# Odyssée User's Guide Version 1.7

Christèle Faure, Yves Papegay

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

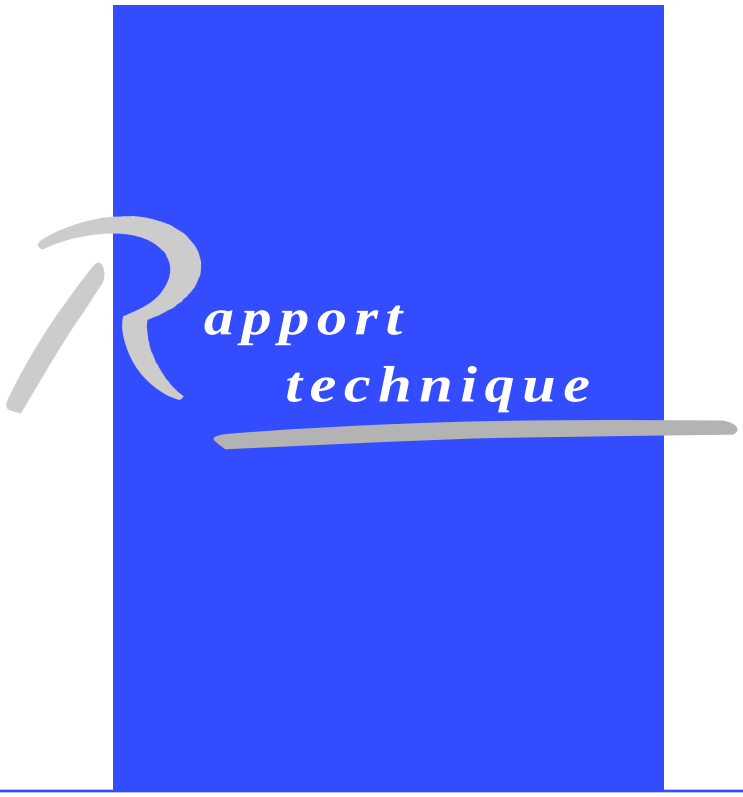
# Odyssée *User's Guide*
# *Version 1.7*

Christèle Faure — Yves Papegay

**N° 0224**

Septembre 1998

THÈME 2




ISSN 0249-0803

# O$\partial y$ssée User's Guide
# Version 1.7

Christèle Faure*, Yves Papegay†

Thème 2 — Génie logiciel
et calcul symbolique
Projet SAFIR

Rapport technique n° 0224 — Septembre 1998 — 81 pages

**Abstract:** O$\partial y$ssée is an *automatic differentiation* processor for FORTRAN 77 code which implements both the forward and reverse mode of automatic differentiation.

This document presents the functionalities of O$\partial y$ssée through an illustrative example. It presents the basic concepts and objects of O$\partial y$ssée, and completely describes the command language of O$\partial y$ssée and its graphical user interface.

**Key-words:** Odyssee, automatic differentiation, computational differentiation, fortran, program transformation

* Email : Christele.Faure@sophia.inria.fr, URL : http://www.inria.fr/safir/WHOSWHO/Christele.Faure
† Email : Yves.Papegay@sophia.inria.fr, URL : http://www.inria.fr/safir/Yves.Papegay

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

# Manuel d'utilisation d'O$\partial y$ssée
# Version 1.7

**Résumé :** O$\partial y$ssée est un logiciel de différentiation automatique de programme FORTRAN 77 qui implémente à la fois les modes direct et inverse de différentiation automatique. Elle est composée d'une boîte à outils, d'un interprète du langage de commande qui permet d'accéder à ces outils et d'une interface graphique à ce langage.

Ce rapport présente les principales fonctionnalités d'O$\partial y$ssée ainsi que les concepts et les objets de base. De plus, il documente complètement le langage de commande d'O$\partial y$ssée et son interface graphique.

**Mots-clés :** Odyssee, différentation automatique, fortran, transformation de programme

# Contents

# Chapter 1

# Introduction

For an introduction to the automatic differentiation theory and techniques one can refer to the proceedings of the main conferences on automatic differentiation: "Automatic Differentiation of Algorithms: Theory, Implementation, and Applications" (see [17]) and "Computational Differentiation: Applications, Techniques, and Tools" (see [1]).

For a survey of the existing tools and their functionalities, visit the web page:
http://www.mcs.anl.gov/Projects/autodiff/index.html
of the Computational Differentiation Project at Argonne National Laboratory.

## What is O$\partial y$ssée?

O$\partial y$ssée is an *automatic differentiation* processor for FORTRAN 77 code:

> Given a set of FORTRAN 77 units that compute a mathematical function $f$, and a list of input variables, it produces FORTRAN 77 subroutines which compute the derivatives of $f$ with respect to those variables.

Compared to other automatic differentiation tools, the key features of O$\partial y$ssée are:

- both the forward and reverse mode of automatic differentiation are implemented,
- *black-box* units (i.e. subroutines or functions whose code is not available) can be accommodated.

In forward mode, O$\partial y$ssée generates an augmented FORTRAN 77 code which computes the function and one directional derivative (tangent) of it (see figure 3.1 for an example). In reverse mode, it generates a FORTRAN 77 code for the computation of a gradient (cotangent) (see figures 3.2 and 4.3 and 4.2 for examples).

O$\partial y$ssée analyzes the FORTRAN 77 code to detect all *active variables*, i.e. variables whose values depend on the variables with respect to which the program is differentiated.

Yet O$\partial y$ssée may also treat calls of subroutines or functions whose code is not available, called *black-box* units. For this purpose, it uses a data base that contains, for each unit, the information on dependencies between its output and input variables.

## Miscellanies

The development of O$\partial y$ssée began in 1991 at the Sophia Antipolis research center of INRIA by members of the SAFIR team. The SAFIR team is a joint team of INRIA[1], CNRS[2] and UNSA[3].

The current version of O$\partial y$ssée is version 1.7. It includes the O$\partial y$ssée engine, an interpreter of the command language and a MOTIF-like graphical user interface. The engine and the interpreter of O$\partial y$ssée have been written in the Objective-CAML[4] language, its interface has been written in the TCL-TK language.

Only the binary versions are distributed outside INRIA and UNSA. It runs on several UNIX platforms. It is available on request without charge for educational and non-profit research and for internal evaluation. For this purpose, or for any other information, the development team of O$\partial y$ssée can be contacted by e-mail at odyssee@sophia.inria.fr.

This document describes completely the command language of O$\partial y$ssée and the menus of its graphical user interface. Other information can be found at the O$\partial y$ssée WWW site at the URL http://www.inria.fr/safir/SAM/Odyssee/odyssee.html.

[1] INRIA is the French national institute for research in computer science and control (http://www.inria.fr/welcome-eng.html)

[2] CNRS is the French national center for scientific research (http://www.cnrs.fr/index.html)

[3] UNSA is the university of Nice Sophia Antipolis (http://www.unice.fr)

[4] http://pauillac.inria.fr/ocaml/index.html

# Chapter 2

# Getting Started

In his document we will consider a sample example and illustrate all the steps to get the numerical values of the derivatives.

Let $\phi$ be the mathematical function of two real variables defined by:

$$\phi : (s,t) \longmapsto \frac{\sqrt{\exp(s^2) + \frac{\sin s}{s}} - \sqrt{\exp(t^2) - \frac{\sin t}{t}}}{1 + \sqrt{\exp(s^2) + \frac{\sin s}{s}} + \sqrt{\exp(t^2) - \frac{\sin t}{t}}}$$

It may be implemented by the FORTRAN 77 code called `head` shown in figure 2.1 page 8 which is a collection of six compilation units. To compute numerical values, this code should be linked to subroutines dedicated to the input of numerical data and to the output of the result. These routines are not presented here, as they are not relevant for differentiation.

In this chapter we show how to compute the numerical value of the gradient of $\phi$ using the tangent and cotangent codes of its implementation within `head`. One must notice that in `head` the *mathematical input variables* $s, t$ are represented by the *computational variables* `i1,i2` and the value of the output $\phi(s,t)$ is stored within the computational variable `o`.

## 2.1 Running O$\partial y$ssée

When O$\partial y$ssée has been correctly installed, typing odyssee or xodyssee are the two ways to run it.

The command odyssee starts the interpreter of the O$\partial y$ssée command language[1]. After the banner, you will get a special prompt `ODYTOP>`: O$\partial y$ssée is waiting for input.

[1] otherwise check if the shell variable PATH is correctly set.

```
double precision function f (t)

double precision t

f = t*t
f = exp(f)
return
end

double precision function g (t)

double precision t

g = 1
if (t .ne. 0.d0) then
   g = sin(t)/t
endif
return
end

subroutine sub1 (x,y)

double precision x,y
double precision f,g

y = f(x) + g(x)
y = sqrt(y)
return
end
```

```
subroutine sub2 (x,y)

double precision x,y
double precision f,g

y = f(x) - g(x)
y = sqrt(y)
return
end

subroutine sub0 (u,v)

common /zz/ zn,zd
double precision u,v,z1,z2,zn,zd

call sub1 (u,z1)
call sub2 (v,z2)
zn = z1 - z2
zd = 1 + z1 + z2
return
end

subroutine head (i1,i2,o)

common /zz/ zn,zd
double precision i1,i2,o,zn,zd

call sub0 (i1,i2)
o = zn/zd
return
end
```

Figure 2.1: Example of FORTRAN 77 Code within the file `test.f`

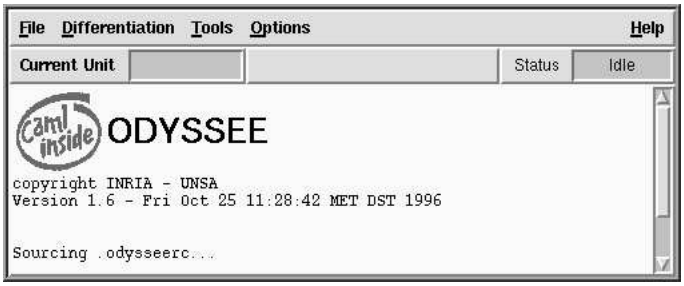



Figure 2.2: Oδyssée initial window

```
                ODYSSEE
        copyright INRIA - UNSA
        Version 1.7

No .odysseerc...
OK
ODYTOP>
```

The message `No .odysseerc...` means that no file *.odysseerc* has been found. This file when it exists is contains the command that configure Oδyssée. If you do not have any, the default values are used for the global variables of Oδyssée (see 7.8 for more details). If the file does exist then the message is `Sourcing .odysseerc....`

If you prefer to start the graphical user interface of Oδyssée, use the command xodyssee[2] and wait until the Oδyssée window appears like in figure 2.2.

In both cases, once Oδyssée has been started, four steps are necessary to get the numerical values of the derivatives:

1. to load the input FORTRAN 77 code into the *internal library*,
2. to differentiate the code,
   (a) to select the differentiation method,
   (b) to select the *head unit* in the internal library,
   (c) to select the the *active input variables* – the variables with respect to which the original code will be differentiated,
3. to save the code generated by Oδyssée in a file,

[2] in case of problems, check that the shell variables DISPLAY and ODYSSEE are correctly set.

```
                ODYSSEE
        copyright INRIA - UNSA
        Version 1.7

Sourcing .odysseerc...
OK
ODYTOP> load test
ODYTOP> libgraph

head +- sub0 +- sub1 +- f
                     +- g
             +- sub2 +- f
                     +- g
```

Figure 2.3: Loading the file `test.f`

4. to write down the driver of the derivative.

We will illustrate all these phases within the next sections.

## 2.2 Loading the original code

The figure 2.3 shows the command *load* which is used to load the file `test.f` shown in the previous figure. When the code is loaded one can look at the its call graph using the command *libgraph*.

## 2.3 Differentiating the code

The figures 2.4(a) and 2.4(b) show the options to the command *diff* necessary to differentiate in direct mode *diff -tl*, respectively reverse mode *diff -cl* the program, which head unit is *diff -h* `head`, with respect to the dummy argument *diff -vars* `i1, i2`. The figures show also the generated program within the library as a call graph.

## 2.4 Saving the derivative units within files

The figures 2.5(a) and 2.5(b) show how to print the derivative code within files using the *getdiffprogram* command. Now the two files `testtl.f, testcl.f` contain all the derivative units.

```
ODYTOP> diff -tl -h head -vars i1 i2
ODYTOP> libgraph

head +- sub0 +- sub1 +- f
                     +- g
             +- sub2 +- f
                     +- g

headtl +- sub0tl +- sub1tl +- ftl
                           +- gtl
                 +- sub2tl +- ftl
                           +- gtl
```

(a) Tangent code

```
ODYTOP> diff -cl -h head -vars i1 i2
ODYTOP> libgraph

head +- sub0 +- sub1 +- f
                     +- g
             +- sub2 +- f
                     +- g


headcl +- sub0 +- sub1 +- f
                        +- g
               +- sub2 +- f
                        +- g
       +- sub0cl +- sub1 +- f
                          +- g
                 +- sub1cl +- f
                           +- fcl
                           +- g
                           +- gcl
                 +- sub2 +- f
                          +- g
                 +- sub2cl +- f
                           +- fcl
                           +- g
                           +- gcl
```

(b) Cotangent code

Figure 2.4: Generation of the derivatives within the library

```
ODYTOP> getdiffprogram headtl testtl
```

(a) Tangent code

```
ODYTOP> getdiffprogram headcl testcl
```

(b) Cotangent code

Figure 2.5: Generation the two files `testtl.f`, `testcl.f`

## 2.5 Writting the two drivers

Now one still have to write down the main program that calls the derivatives.

The figures 2.6(a) and 2.6(b) show that the derivatives associated to the dummy arguments `i1`, `i2`, `o` of `head` are `i1ttl`, `i2ttl`, `ottl` in the tangent code and `i1ccl`, `i2ccl`, `occl` in the cotangent code. But one must notice that if `i1`, `i2` are input dummy arguments and `o` is the output. In the tangent code, `i1ttl, i2ttl` have the same status as `i1`, `i2` and `ottl` has is output as well as `o`. In the cotangent mode (due to the transposition), `occl` becomes input and `i1ccl`, `i2ccl` are input and output.

One must notice that a file *odyparam.inc* (see 5.2.3 for more details) is automatically included within the generated code. In this exemple, this file is empty but even though has to be created.

In the figures 2.7(a) and 2.7(b) we show the main program that computes the gradient $\partial i1/\partial o, \partial i2/\partial o$ using the tangent code or the cotangent code. In the tangent code, the direction $i1ttl, i2ttl$ is initialized to $1, 0$ to get the first partial derivative, and to $0, 1$ to get the second. In the cotangent code, the dual variables $i1ccl, i2ccl$ are initialized to zero, and $occl$ is initialize to 1.

## 2.6 Compiling, linking and running the derivatives

The definitions of `maintl, mancl` have been written within two files: `maintl.f, mancl.f`. Then one has to compile and link the files together (see 2.8(a), 2.8(b)) and to execute the compiled codes to get the value of the gradient for a specific value of the original inputs of the program $i1, i2$ (in our example $i1 = 1.5$ and $i2 = -0.5$).

One must notice that the original code of `test.f` is needed for the construction of the cotangent compiled code, but not for the tangent code in this example.

```
ODYTOP> getunit headtl
COD Compilation unit : headtl

      SUBROUTINE headtl (
     :           i1, i2, o,
     :           i1ttl, i2ttl, ottl)

      IMPLICIT NONE
      INCLUDE 'odyparam.inc'
      DOUBLE PRECISION i1
      DOUBLE PRECISION i2
      DOUBLE PRECISION o
      DOUBLE PRECISION i1ttl
      DOUBLE PRECISION i2ttl
      DOUBLE PRECISION ottl
      EXTERNAL sub0
      EXTERNAL sub0tl
      DOUBLE PRECISION znttl
      DOUBLE PRECISION zdttl
      COMMON /zzttl/znttl, zdttl
      DOUBLE PRECISION zn
      DOUBLE PRECISION zd
      COMMON /zz/zn, zd
```

(a) Tangent code

```
ODYTOP> getunit headcl
COD Compilation unit : headcl

      SUBROUTINE headcl (
     :           i1, i2, o,
     :           i1ccl, i2ccl, occl)

      IMPLICIT NONE
      INCLUDE 'odyparam.inc'
      DOUBLE PRECISION o
      DOUBLE PRECISION i1
      DOUBLE PRECISION i2
      DOUBLE PRECISION i1ccl
      DOUBLE PRECISION i2ccl
      DOUBLE PRECISION occl
      EXTERNAL sub0
      EXTERNAL sub0cl
      DOUBLE PRECISION zn
      DOUBLE PRECISION zd
      COMMON /zz/zn, zd
      DOUBLE PRECISION znccl
      DOUBLE PRECISION zdccl
      COMMON /zzccl/znccl, zdccl
```

(b) Cotangent code

Figure 2.6: Declaration within the derivatives of `head`

```
      program maintl

      DOUBLE PRECISION i1, i1ttl
      DOUBLE PRECISION i2, i2ttl
      DOUBLE PRECISION o, ottl

C Initialization of the function
      i1 = 1.5
      i2 = -0.5
C Initialization of the first direction
      i1ttl = 1.
      i2ttl = 0.
      call headtl (i1, i2, o, i1ttl, i2ttl, ottl)
      di1do = ottl
C Initialization of the second direction
      i1ttl = 0.
      i2ttl = 1.
      call headtl (i1, i2, o, i1ttl, i2ttl, ottl)
      di2do = ottl
C Printing of the derivative values
      write (6,*) "di1/do = ", di1do
      write (6,*) "di2/do = ", di2do
      end
```

(a) Tangent code

```
      program maincl

      DOUBLE PRECISION i1, i1ccl
      DOUBLE PRECISION i2, i2ccl
      DOUBLE PRECISION o, occl

C Initialization of the function
      i1 = 1.5
      i2 = -0.5
C Initialization of the dual variables
      occl = 1.
      i1ccl = 0.
      i2ccl = 0.
      call headcl (i1, i2, o, i1ccl, i2ccl, occ
      di1do = i1ccl
      di2do = i2ccl
C Printing of the derivative values
      write (6,*) "di1/do = ", di1do
      write (6,*) "di2/do = ", di2do
      end
```

(b) Cotangent code

Figure 2.7: Writtin the drivers

```
$ f77 -o testtl maintl.f testtl.f
maintl.f
 MAIN maintl:
testtl.f:
        headtl:
        sub0tl:
        sub2tl:
        gtl:
        ftl:
        sub1tl:
Linking:
$ testtl
 di1/do =    0.649161
 di2/do =     8.69188E-02
```

(a) Tangent code

```
$ f77 -o testcl maincl.f testcl.f test.f
maincl.f
 MAIN maincl:
testcl.f:
        headcl:
        sub0cl:
        sub2cl:
        gcl:
        fcl:
        sub1cl:
test.f:
        head:
        sub0:
        sub1:
        sub2:
        f:
        g:
        s3:
        s4:
        s5:
        s6:
Linking:
$ testcl
 di1/do =    0.649161
 di2/do =     8.69188E-02
```

(b) Cotangent code

Figure 2.8: Compilling, linking and running

# Chapter 3

# Elementary algorithms

Odyssée is an automatic differentiation tool developed at INRIA that differentiates Fortran-77 units. If a function is implemented as a set of units, Odyssée is able to differentiate it as a whole with respect to the inputs specified by the user. From this set of units, Odyssée generates a new set of units that computes the derivatives. In Odyssée, the two modes of automatic differentiation have been implemented. In direct mode Odyssée uses the tangent linear algorithm to generate a Fortran-77 code that computes one directional derivative (i.e. the product of the jacobian matrix by one direction). In reverse mode, the code generated by Odyssée computes the cotangent code (equivalent to hand written adjoint codes) which is the product of a vector by the transposed jacobian matrix.

## 3.1 The process of differentiation in Odyssée

The process of differentiation consists of four phases:

1. preparation of the code (see the *preprocess* command),
2. interprocedural analysis of the program (see the *computeibasis* command),
3. differentiation of each routine,
4. simplification of the code.

During the first phase, the system modifies the original units and puts them in a suitable form for differentiation. It rewrites the nondifferentiable operators `abs`, `min` into `if`-statements. The `function` statements are also in-lined into the code. Odyssée splits expressions into equivalent binary expressions by introducing intermediate variables (see [9]). This is based on the fact that: if the code is formed of binary expressions, then the complexity of the calculation of the function and its derivative is less than four or five times the one of the function alone (this depends on the differentiation mode).

During the second phase, O$\partial$yssée makes an interprocedural analysis of the program, which results in a dependency graph between the input/output variables of each unit. These dependencies are stored in a data basis, which is filled by the differentiator for the available subroutines. For the subroutines whose code is not supplied, the user has to fill first the data basis, by declaring what are the arguments of the subroutine and which are the input or/and output variables. This information is enough for O$\partial$yssée to make a consistent analysis of the overall program. At that point, the system is able to propagate the active variables from the head-unit to all the other units. Thus the active inputs of each subroutine are known and O$\partial$yssée can differentiate them independently.

In the third phase, the unit by unit differentiation of the program is done, according to the following two rules. First, each subroutine of the program is differentiated with respect to its input variables, and not with respect to the input variables of the head-unit. Secondly, the differentiation is *maximal* in the sense that a subroutine is differentiated with respect to the maximum set of input variables appearing in every `call` statement. This allows O$\partial$yssée to generate only one subroutine for each unit of the original program. The methods used to differentiate a subroutine are described in the next section.

The fourth phase aims at simplifying the resulting code. The algebraic expressions are simplified in the same way as in Computer Algebra Systems except that the arguments of sums and products cannot be reordered modulo associativity and commutativity. The dead code is suppressed, which is very important in the reverse mode of differentiation, because the system generates too many `saving` instructions.

## 3.2 Differentiation of one assignment

In this section, we will show the derivative of the following assignment:

```
Z = X * Y**2.
```

In direct mode, the generated code computes the product of the jacobian by one direction. For example, the derivative of the previous instruction is:

```
dZ = Y**2 * dX + 2*X*Y*dY.
```

If one wants to get the gradient of this instruction he has to call it twice with

1. $dX = 1., dY = 0.$ to get $dZ = \frac{\partial Z}{\partial X}$,
2. $dX = 0., dY = 1.$ to get $dZ = \frac{\partial Z}{\partial Y}$.

In reverse mode, the generated code computes the product of a vector by the transposed jacobian. For example, the derivative of the previous instruction is:

```
dX = dX + Y**2*dZ
dY = dY + 2*X*Y*dZ
dZ = 0.
```

If one wants to get the gradient of this instruction, he has to call it one time with

$$dZ = 1., dX = 0., dY = 0.$$

and will get

1. $dX = \frac{\partial Z}{\partial X}$,
2. $dY = \frac{\partial Z}{\partial Y}$,
3. $dZ = 0$.

## 3.3 Differentiation of a routine

In this section, we will show the derivative of the following simple routine :

```
COD Unite : s3
COD Dummys:  n x y z
COD IN   dummys:  x y z
COD OUT  dummys:  x z
COD Dependencies between IN and OUT:
COD x <--   x
COD z <--   x y z

SUBROUTINE s3 (n, x, y, z)

DIMENSION x(n), y(n), z(n)

x(1) = 0.
i = 1
DO WHILE (i.LE.n )
   IF (x(i).LT.y(i)) THEN
     z(i) = y(i)
   ELSE
     z(i) = x(i)
   END IF
   i=i+1
END DO
RETURN
END
```

In direct mode, a sequence of instructions is differentiated by differentiating each original instruction in the same order. The figure 3.1 shows the *tangent code* generated with the command : *diff -tl -vars y -h s3*.

```
COD This file has been generated by Odyssee  1.7
COD Program : s3tl

COD Compilation unit : s3tl
COD Derivative of unit : s3
COD Dummys:  n x y z
COD Active IN   dummys:  y
COD Active OUT  dummys:  z
COD Dependencies between IN and OUT:
COD z <--   y


SUBROUTINE s3tl (n, x, y, z, yttl, zttl)

IMPLICIT NONE
INTEGER i
INTEGER n
REAL zttl(n)
REAL x(n)
REAL y(n)
REAL z(n)
REAL yttl(n)

x(1) = 0.
i = 1
DO WHILE (i.LE.n )
   IF (x(i).LT.y(i)) THEN
     zttl(i) = yttl(i)
     z(i) = y(i)
   ELSE
     zttl(i) = 0.
     z(i) = x(i)
   END IF
   i = 1+i
END DO
RETURN
END
```

Figure 3.1: Tangent code of `s3`

In reverse mode, O∂yssée generates the cotangent code. The cotangent code of each original routine is divided into two parts : the *direct part* which computes the original function and stores all the modified values, and the *reverse part* which restores the correct values of the source variables and computes the derivatives.

Two algorithms have been implemented in reverse mode for the differentiation of a sequence of instructions. One is general and can be applied on any code, we call it the "*flow reversal*" algorithm but with this algorithm the generated code is not easily readable. The second one is devoted to codes without **goto, exit, return** because it reverses the code syntactically and is called here "*syntax reversal*". In this section about basics of O∂yssée we do not show the "*flow reversal*" algorithm but it is described in appendix 4.1.

We show the *cotangent code* generated from the simple previous example with the "syntax reversal" using the command *diff -cl -vars y -h s3* in figure 3.2.

The "syntax reversal" algorithm deals with the program syntactically which means that a `DO i=1,n` loop in the direct part will be reverted as a `DO i=n,1` loop. In the direct part (see figure 3.2(b)), the generated code executes the original instructions and stores the initial values of the modified variables within local (static) variables. In the reverse part (see figure 3.2(b)), the generated code restores the corresponding values of the variables and then computes the correct derivatives.

As in the implementation of this algorithm we use static save variables. Sometimes, O∂yssée does not know the number of steps of the loops and then uses *odysseemax* as the size of the save array. The user has then to define the correct value of this parameter with an include file called *odyparam.inc*.

```
SUBROUTINE s3cl (n, x, y, z, yccl, zccl)

IMPLICIT NONE
INCLUDE 'odyparam.inc'
INTEGER i
INTEGER n
REAL x(n)
REAL y(n)
REAL z(n)
REAL yccl(n)
REAL zccl(n)


REAL save1
INTEGER save2
INTEGER coun3
LOGICAL test5(odysseemax)
REAL save6(odysseemax)
REAL save7(odysseemax)
INTEGER save8(odysseemax)
INTEGER indi9
```

(a) Declarations

```
C
C Trajectory
C
save1 = x(1)
x(1) = 0.
save2 = i
i = 1
coun3 = 0
DO WHILE (i.LE.n )
   coun3 = 1+coun3
   test5(coun3) = x(i).LT.y(i)
   IF (test5(coun3)) THEN
     save6(coun3) = z(i)
     z(i) = y(i)
   ELSE
     save7(coun3) = z(i)
     z(i) = x(i)
   END IF
   save8(coun3) = i
   i = 1+i
END DO
C
C Transposed linear forms
C

indi9 = coun3
DO coun3 = indi9, 1, -1
   i = save8(coun3)
   IF (test5(coun3)) THEN
     z(i) = save6(coun3)
     yccl(i) = yccl(i)+zccl(i)
     zccl(i) = 0.
   ELSE
     z(i) = save7(coun3)
     zccl(i) = 0.
   END IF
END DO
i = save2
x(1) = save1
RETURN
END
```

(b) syntax reversal

Figure 3.2: Cotangent code of `s3` (syntax reversal)

# Chapter 4

# Sophisticated algorithms

In this chapter we describe the usage of two algorithms related to the reverse mode.

## 4.1 Flow reversal algorithm

In the previous section, we have shown cotangent code generated using the "syntax reversal" algorithm. In this section we show on the same sample example the "*flow reversal*" algorithm.

The "flow reversal" algorithm deals with the program as an execution pile of basic blocks (straight line pieces of program). In order to manage this pile and the values of the trajectory, some dynamic structures have been defined.
For example, the simple code `s3` can be divided within four basic blocks shown in figure 4.1. The execution of `s3` can be described as the sequence $B_1((B_2 or B_3)B_4)^n$ which means that $B_1$ is necessarily executed, then $B_2$ or $B_3$ followed by $B_4$ is executed $n$ times.

To get the order of execution of the derivative of the different blocks in the reverse order, one has only to pop from the execution pile the block indexes. That is the general method we have implemented within the "flow reversal" algorithm.

The figure 4.3 shows the *cotangent code* of `s3` generated by the following command : *diff -cl -goto -vars y -h s3*.

```
x(1) = 0.
i = 1
```

(a) $B_1$

```
z(i) = x(i)
```

(b) $B_2$

```
z(i) = y(i)
```

(c) $B_3$

```
i = i + 1
```

(d) $B_4$

Figure 4.1: Basic blocks from `s3`

```
SUBROUTINE s3cl (n, x, y, z, yccl, zccl)

IMPLICIT NONE
INTEGER i
INTEGER n
REAL x(n)
REAL y(n)
REAL z(n)
REAL yccl(n)
REAL zccl(n)

INTEGER i0
INTEGER p
INTEGER kk
INTEGER block
INTEGER maxp
INTEGER k(4)

p = -1
DO i0 = 1, 4
   k(i0) = 0
END DO
```

(a)

Figure 4.2: Declaration within the cotangent code of `s3`

```
C
C Trajectory
C
p = 1+p
CALL setblock(1, p, 1)
k(1) = 1+k(1)
kk = k(1)
CALL setsave(1, kk, 8, x(1), 1)
x(1) = 0.
CALL setsave(2, kk, 4, i, 1)
i = 1
DO WHILE (i.LE.n )
   IF (x(i).LT.y(i)) THEN
     p = 1+p
     CALL setblock(1, p, 4)
     k(4) = 1+k(4)
     kk = k(4)
     CALL setsave(4, kk, 8, z(i), 1)
     z(i) = y(i)
   ELSE
     p = 1+p
     CALL setblock(1, p, 2)
     k(2) = 1+k(2)
     kk = k(2)
     CALL setsave(3, kk, 8, z(i), 1)
     z(i) = x(i)
   END IF
   p = 1+p
   CALL setblock(1, p, 3)
   k(3) = 1+k(3)
   kk = k(3)
   CALL setsave(5, kk, 4, i, 1)
   i = 1+i
END DO
GOTO 5000
```

(a) Direct part

```
C
C Transposed linear forms
C
 5000 CONTINUE
maxp = p
DO p = maxp, 0, -1
   CALL getblock(1, p, block)
   kk = k(block)
   k(block) = -1+k(block)
   IF (block.EQ.1) THEN
     CALL getsave(2, kk, 4, i, 1)
     CALL getsave(1, kk, 8, x(1), 1)
   ELSE
     IF (block.EQ.2) THEN
       CALL getsave(3, kk, 8, z(i), 1)
       zccl(i) = 0.
     ELSE
       IF (block.EQ.3) THEN
         CALL getsave(5, kk, 4, i, 1)
       ELSE
         CALL getsave(4, kk, 8, z(i), 1)
         yccl(i) = yccl(i)+zccl(i)
         zccl(i) = 0.
       END IF
     END IF
   END IF
END DO
END
```

(b) Reverse part

Figure 4.3: Cotangent code of `s3`

In the direct part (see figure 4.3(a)), the generated code executes the original instructions and stores the sequence of block indexes (calling setblock) as well as the initial values of the modified variables (calling setsave) in a pile.

In the reverse part (see figure 4.3(b)), the generated code restores the current block index from the pile (calling getblock), tests its value and restores the corresponding values of the variables (calling getsave) and then computes the correct derivatives.

The *setblock* routine has got three arguments `i, p, block` where `i` is the index of the routine, `p` is the index of the block in the routine, `block` is the index of the block and stores the index of the block. The *getblock* routine uses the same three arguments and restores the index of the current block.

The *setsave* routine has got five arguments `i, p, byte, val, size` where `i` is the index of the save within the program, `p` is the index of the current save, `byte` is the number of bytes, `val` is a pointer to the value, `size` is the size of the value, and stores `size*byte` bytes from the pointer `val`. The *getsave* routine uses the same arguments and restores the current value.

An implementation of these routines has been written in C and is distributed with O$\partial$*y*ssée. It uses dynamic allocation in order to implement one pile for the block indexes and one pile for the saves. But one can check other implementations easily.

In the generated code, `p` is the current number of executed blocks, `k` is an array of integers where `k(i)` is the number of iterations of the block `i` , `kk` is the number of iteration of the current block. `maxp` it the maximum number of block to be executed in the reverse part and `block` is the index of the current block whithin the reverse part.

## 4.2 Checkpointing facility

The two algorithms that implement the reverse mode in O$\partial$*y*ssée store all the modified variables. Obviously, it stores all the modified variables at each step of the loop.

A strategy for mixing storage and recomputation of the variables modified within a loop has been studied. It has been proven that when the number of steps of the loop to be stored (called *checkpoints*) is chosen, a schedule optimal in term of recomputation of the body of the loop can be found (see [18, 19]).

We have implemented such a checkpointing facility. This implementation allows the user to chose the number of checkpoints and then give the optimal schedule in execution time using one predefined routine called *compute_ q* (distributed with O$\partial$*y*ssée). Within O$\partial$*y*ssée, it can only be applied to explicit loops whose body is a unique call to a routine (called here the inner routine). For example, the code of `s3` can also be rewritten as shown in figure 4.4 to fit this structure.

In order to use this algorithm the routine *optimal_ reverse* from the file *opt_ rev* must be loaded. This file is distributed with O$\partial$*y*ssée. Let say that one differentiates `s5` in reverse mode using the checkpointing facility, the command to be applied is: *diff -cl -opt s6 -vars y -h s5* . One will get the standard derivative for `s6` (see figure 4.5(b)), but a specific derivative for `s5`. The derivative of `s5` is implemented within two subroutines : `s5cl` (see figure 4.5(a))

```
SUBROUTINE s5 (n, x, y, z)

DIMENSION x(n), y(n), z(n)

x(1) = 0.
i = 1
DO WHILE (i.LE.n )
   call s6 (i, n, x, y, z)
END DO
RETURN
END
```

(a)

```
SUBROUTINE s6 (i, n, x, y, z)

DIMENSION x(n), y(n), z(n)

IF (x(i).LT.y(i)) THEN
   z(i) = y(i)
ELSE
   z(i) = x(i)
END IF
i=i+1
END
```

(b)

Figure 4.4: `s3` rewritten in the correct format

and `opts6cl` (see figure 4.6). In this section we call *register* the vector of all output variables of the inner routine (`s6` in our example). Each variable of the register is stored within a variable which prefix is *regi*. Its size is *nb_reg* times the initial size. `s5cl` is standard except for the treatment of the `DO while` loop :

1. in the direct part, the output variables of `s6 i, z` are stored within the first *register* `regi6(0),regi7(0,.)` before the loop and the loop is executed (without storing any thing else),
2. in the reverse part, the derivative of the whole `DO`-loop is replaced by a call to `opts6cl`.

`opts6cl` is the derivative of the `DO`-loop and uses the checkpointing method in order not to store the values modified by `s6` at all the steps of the loop, but only at `nb_reg` steps.

In the generated *cotangent code*, three parameters have to be set by the user to correct value: `nb_reg, p_max, odyn`. This can be done using the standard include file called *odyparam.inc* which is not generated by O*dyssée*.

The two parameters that depend on the checkpointing method are:

1. *nb_reg* is the number of checkpoint (i.e. the number of registers to be stored). It can be set to any positive integer.
2. *p_max* is the size of the pile used by the algorithm instead of a recursive routine. It must be set to the maximum number of iterations of the loop.

In this example, a parameter *odyn* appears. This parameter is due to the declaration of the global variables where the registers are stored. As `n` is an argument of the function, it can not be used as the size of a global parameter. The parameter `odyn` stands for parameter whose value is `n`.

```
SUBROUTINE s5cl (n, x, y, z, yccl, zccl)

IMPLICIT NONE
INCLUDE 'odyparam.inc'
INTEGER i
INTEGER n
REAL x(n)
REAL y(n)
REAL z(n)
REAL yccl(n)
REAL zccl(n)

INTEGER n_max
REAL save1
INTEGER save2
INTEGER coun3
INTEGER nnn1
COMMON /regi/regi6, regi7
REAL regi6(0:nb_reg)
REAL regi7(0:nb_reg,odyn)
C
C Trajectory
C
save1 = x(1)
x(1) = 0.
save2 = i
i = 1
regi6(0) = i
DO nnn1 = 1, n
   regi7(0,nnn1) = z(nnn1)
END DO
coun3 = 0
DO WHILE (i.LE.n )
   coun3 = 1+coun3
   CALL s6(i, n, x, y, z)
END DO
n_max = -1+coun3
C
C Transposed linear forms
C
CALL opts6cl(n_max, i, n, x, y, z, yccl, zccl)
i = save2
x(1) = save1
RETURN
END
```

(a)

```
SUBROUTINE s6cl (i, n, x, y, z, yccl, zccl)

IMPLICIT NONE
INTEGER n
INTEGER i
REAL x(n)
REAL y(n)
REAL z(n)
REAL yccl(n)
REAL zccl(n)

LOGICAL test2
REAL save3
REAL save4
INTEGER save5
C
C Trajectory
C
test2 = x(i).LT.y(i)
IF (test2) THEN
  save3 = z(i)
  z(i) = y(i)
ELSE
  save4 = z(i)
  z(i) = x(i)
END IF
save5 = i
i = 1+i
C
C Transposed linear forms
C
i = save5
IF (test2) THEN
  z(i) = save3
  yccl(i) = yccl(i)+zccl(i)
  zccl(i) = 0.
ELSE
  z(i) = save4
  zccl(i) = 0.
END IF
END
```

(b)



Figure 4.5: Derivative of `s5`, `s6`

```
SUBROUTINE opts6cl (n_max, i, x, y, z,
                    yccl, zccl)

IMPLICIT NONE
INCLUDE 'odyparam.inc'
INTEGER i
INTEGER n
REAL x(n)
REAL y(n)
REAL z(n)
REAL yccl(n)
REAL zccl(n)

INTEGER ipile
INTEGER k
INTEGER i1
INTEGER n_max
INTEGER p
INTEGER q
INTEGER nnn1
EXTERNAL compute_q

INTEGER pile1(p_max)
INTEGER pile2(p_max)
COMMON /regi/regi6, regi7
REAL regi6(0:nb_reg)
REAL regi7(0:nb_reg,odyn)
```

(a)

```
pile1(1) = n_max
pile2(1) = 0
ipile = 1
DO WHILE (ipile.GT.0 )
   p = pile1(ipile)
   i1 = pile2(ipile)
   ipile = ipile-1
   IF (p.LT.0) THEN
     WRITE (6, *) 'Error : p negatif'
     STOP
   END IF
   i = regi6(i1)
   DO nnn1 = 1, n
      z(nnn1) = regi7(i1,nnn1)
   END DO
   IF (p.EQ.0) THEN
      CALL s6cl(i, n, x, y, z, yccl, zccl)
   ELSE
     CALL compute_q(nb_reg-i1, p, q)
     IF (q.EQ.0) THEN
    WRITE (6, *) 'Error : q=0'
    STOP
     END IF
     DO k = 1, q
        CALL s6(i, x, y, z)
     END DO
     regi6(i1+1) = i
     DO nnn1 = 1, n
        regi7(i1+1,nnn1) = z(nnn1)
     END DO
     IF ((ipile+2).GT.p_max) THEN
    WRITE (6, *)
     : 'Error : pile pleine, modifier p_max'
    STOP
     END IF
     ipile = ipile+1
     pile1(ipile) = q-1
     pile2(ipile) = i1
     ipile = ipile+1
     pile1(ipile) = p-q
     pile2(ipile) = i1+1
   END IF
END DO
RETURN
END
```

(b)



Figure 4.6: Derivative of the DO-loop

# Chapter 5

# Frequently asked questions

Some of those remarks which are a bit strict have changed since the previous version of Odyssée and will be modified to make Odyssée easier to use within the next version.

## 5.1 On the original code

### 5.1.1 Include files

The include files are read from the global Odyssée variables named `include_path`.

### 5.1.2 Modification of the source code

Before differentiation, the system modifies the copy of the original units. The modified routines appear in the Odyssée library.

### 5.1.3 Active input variables

The only active input variables with respect to which a code can be differentiated are arguments or global variables.

### 5.1.4 Active output variables

The user cannot specify the active output variable he wants the generated code to compute. All the output variables which depend on the active input variable will be differentiated.

### 5.1.5 Global variables within common blocks

Odyssée checks whether or not all the `common blocks` are split into exactly the same global variables (same name, FORTRAN 77 type and size) in all the units. It also checks if two global

variables from two different common blocks, or a global variable and a common block do not have the same name. If those restrictions are not verified, an error message is generated.

### 5.1.6 `equivalence` instruction

Equivalences between symbols are not correctly dealt with, the user has to verify the generated code.

### 5.1.7 `read, write` instruction

`read, write` instructions are considered as constant and do not have any associated derivative. If you have some of them in your code, you may add the desired derivatives.

### 5.1.8 The jumps in reverse mode

The "*syntax reversal*" algorithm does not treat codes with `goto`, `return` in the middle of the code. O$\partial$yssée generates an error message if they appear in the original code. The other algorithm in reverse mode refered to as "*flow reversal*" is able to differentiate a code with those instructions.

## 5.2 On the generated code

### 5.2.1 Include files

The include files are in-lined in the differentiated code.

### 5.2.2 Declaration of variables

We have chosen to declare all the variables in the differentiated code. Even if in the original code some *implicit* declarations are used they are no more implicit in the derivative.

### 5.2.3 *O$\partial$yssée parameters* and odyparam.inc

O$\partial$yssée uses some parameters that do not exist in the original code in order to set the size of the save or init arrays. We have chosen not to define their value in order to force the user to declare them.

*odysseemax* is used when O$\partial$yssée knows nothing about the size of a loop. It has to be set to the maximum number of iterations of the loop.

When O$\partial$yssée generates a copy of an array whose size is `n`, this copy has got the same size. But in FORTRAN 77 the size of array variables must be known at compile time. Therefore O$\partial$yssée can not declare the size of such a copy to be `n` if `n` is global or argument. It creates the new name *odyn* which has to defined as a parameter with the value of `n`.

Using the checkpointing algorithm (in reverse mode) Odyssée uses two other parameters *p_max* and *nb_reg*.

In each derivative routine Odyssée includes the *odyparam.inc* file. This file is entended to group all the declarations of all the Odyssée parameters, and are not generated by the system.

### 5.2.4 Black box routines

Odyssée is able to differentiate a code with *black box* routines in it if the user suplies their IN/OUT definition in an information file, but their derivative are not generated by the system. The user has to write the derivatives accordingly to the `Call` generated within the differentiated units. This can be done using finite differences, reusing the original unit if it is linear or self adjoint.

# Chapter 6

# The O∂yssée Concepts

## 6.1 O∂yssée and the Fortran 77 Code

**A compilation unit** is a part of a file of FORTRAN 77 code that can be compiled separately. Therefore a compilation unit is one of the following FORTRAN 77 entities: a subroutine, a function, a program, or a block data. O∂yssée differentiate only subroutines, or functions.

**The internal library** is the structure where O∂yssée stores compilation units:

- the units loaded by O∂yssée,
- the units generated by O∂yssée (results of a differentiation process).

When a new session of O∂yssée starts, the internal library is empty.

**The head unit** is the routine chosen by the user that computes the mathematical function to be differentiated. Usually, a mathematical function is represented as a collection of compilation units. In order to differentiate it, the user has to give O∂yssée the name of the head unit.

## 6.2 O∂yssée and the Dependencies

**The variables** of a compilation unit is the collection of all the variables appearing in this unit. It includes (see figure 6.1 page 37):

- formal parameters or dummy arguments : the arguments of subroutines and functions,
- global variables : parts of common blocks,

- local variables : other variables.

One must notice that O$\partial$yssée considers the function name as a variable, as it is attached to the output value of the corresponding function.

**The input and output variables** of a compilation unit are global or dummy variables.

- *input variables* are the non-local variables which should have a value attached to before the unit could be run,
- *output variables* are the non-local variables which are modified when the unit is run.

Note that some variables may be at the same time input and output variables (see figure 6.2 page 37).

**The dependencies** are defined between the input and the output variables of a compilation unit: an output variable depends on an input variable when it is necessary to know the value of the input variable to be able to compute the output variable. For each compilation unit, it is possible to define a function $f$ from the set $\mathcal{O}$ of its output variables to the set $\mathcal{P}(\mathcal{I})$ of the subsets of the set $\mathcal{I}$ of its input variables:

$$f \quad : \quad \begin{array}{ccc} \mathcal{O} & \longrightarrow & \mathcal{P}(\mathcal{I}) \\ o & \longmapsto & \{i_1, \ldots, i_n\} \end{array}$$

such as $i \in f(o)$ if and only if it is necessary to know the value of $i$ to compute the value of $o$.

In fact, it is even possible to define two such functions, one concerning the *control-dependencies*, the other concerning the *value-dependencies*. The following example shows the difference between control-dependencies and values-dependencies:

> In the FORTRAN 77 instruction
>
> ```
> if (x.gt.0.) then
>   z = y
> end if
> ```
>
> the variable `z` is control-dependent on the variable `x` and value-dependent on the variable `y`.

The only dependencies considered by O$\partial$yssée are the value-dependencies.

**The information base** is the structure where O$\partial$yssée stores, for each unit all the necessary information on the non-local variables and on the dependencies between them:

- its input dummy variables,
- its input global variables,

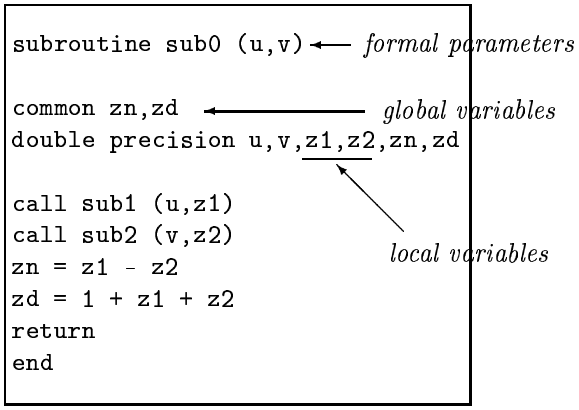



Figure 6.1: different variables

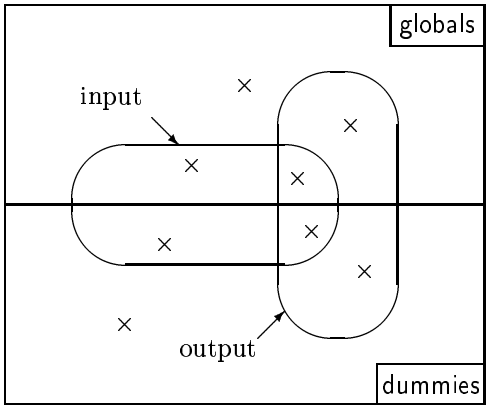



Figure 6.2: Input / output variables

- its output dummy variables,
- its output global variables,
- the dependencies between its output variables and its input variables.

The entry in this base for each unit is filled with default values when the unit is loaded and is really computed when the system performs the analyse of the code. These entries may also be changed by the user through an *information file* (see section 7.4).

**An active variable** is a variable that depends on the variables with respect to which the mathematical function must be differentiated (subset of input variables of the head unit).

**The active information base** is a restriction of the information base that concerns only active variables. The active information base is also attached to a diff computation. It does not exist before the first differentiation and is re-computed at any evaluation of a diff command. Only the information contained in the current active information base may be accessed.

## 6.3 O$\partial y$ssée and the File System

O$\partial y$ssée reads or writes information through several kinds of text files. O$\partial y$ssée considers five different types for files: fortran, include, batch, ibasis and odyssee and a special file: .odysseerc.

Three global variables control each type of file T: T_ext for the name extension, T_dir for the directory where the files have to be written by O$\partial y$ssée, and T_path for a list of path from where O$\partial y$ssée reads the files.

- The fortran files are the files containing FORTRAN 77 code read as input of a load command or generated by a diff computation and written by a getlibrary, a getunit, a getprogram or a getdiffprogram.
- The include files are the files included by the FORTRAN 77 code to be loaded. Note that they are in-lined in the code of the compilation units in the library.
- The batch files contain valid O$\partial y$ssée commands and are executed through the loadbatch command.
- The ibasis files contain O$\partial y$ssée commands that modify the information base (setdummys, setglobals, setinout, setabstract). If an ibasis file of name ⟨*unit*⟩_BI.⟨*extension*⟩ where ⟨*extension*⟩ is the value of the O$\partial y$ssée global variable ibasis_ext exists, it is automatically loaded by O$\partial y$ssée during a diff command applied to ⟨*unit*⟩.
- The odyssee files are the files where O$\partial y$ssée writes information about the internal library like in the listunits, libgraph or getgraph command.

- The .odysserc file is a special batch-file automatically loaded at the beginning of any session of Oâyssée. It can also be reloaded by the loadodysserc command. It is generally used to configure or customize Oâyssée.

# Chapter 7

# The O∂yssée Command Language

## 7.1 Command names

The O∂yssée *character set* consists of the 52 lower case and upper case letters, the 10 digits and the special characters of the table 7.1.

| | |
|---|---|
| \ | backslash |
| ␣ | blanck |
| - | dash |
| . | dot |
| " | double quotes |
| # | sharp |
| / | slash |
| _ | underscore |
| ( | left parenthesis |
| ) | right parenthesis |

Table 7.1: special characters

Tokens of the O∂yssée command language are *strings*, *identifiers*, *lists* and *numbers*:

- A string is a sequence of any printable characters enclosed in double quotes.
- An identifier is a letter, a dot or a slash followed by zero or more letters, digits, dots, dashes, underscores or slashes. true and false are the identifiers corresponding to the two boolean constants. These constants, the command identifiers and the option names are reserved words of the language. They are listed in the table 7.2.
- A list is a sequence of tokens separated by white spaces, for clarity those tokens may be enclosed between parentheses.

| commands | | | others |
|---|---|---|---|
| clear | getlibrary | preprocess | -cl -cotangent |
| computeibasis | getprogram | printunit | -goto |
| copyright | getsymbols | quit | -h -head |
| diff | getunit | remove | -o -output |
| exit | getvar | setabstract | -opt -optloops |
| getabstract | help | setdummys | -split |
| getactiveinout | libgraph | setglobals | -tl -tangent |
| getcommons | listunits | setinout | -vars -variables |
| getdependency | load | setvar | |
| getdiffprogram | loadbatch | shell | |
| getgraph | loadibasis | slice | false |
| getibasis | loadodysseerc | version | true |
| getinout | makeblock | | |

Table 7.2: reserved words

- A number is a signed integer. On most computer architectures, it may have any value in the range $-2^{30} \ldots (2^{30} - 1)$.

An input of the O$\partial$yssée interpreter consists of a single statement usually referred to as a *command line*. However a line, i.e. a sequence of characters followed by a newline character, may contain an incomplete statement. And a statement may be continued from line to line by preceding the newline character by a backslash. In this case, both the backslash and the newline are ignored by O$\partial$yssée.

When the parser of O$\partial$yssée "sees" a sharp character anywhere on a line, any following character until the next newline is treated as a *comment*.

A command line is a sequence of tokens separated by white spaces – a *white space* is either a blank or a tabulation. The first token of a command line should be a valid *command name*. Each command has its own syntax described in one of the following sections.

Except in a few cases – namely when reading arguments of an help command, O$\partial$yssée does not distinguish between an identifier and a string.

## 7.2 The Internal Library Related Commands

### 7.2.1 Modifying the Internal Library

**clear**
Resets the internal library to the empty set.

**load** ⟨*file*$_1$⟩ [⟨*file*$_2$⟩ ... ⟨*file*$_N$⟩]
Loads the compilation units from the specified FORTRAN 77 file(s) in the internal library.

If the name ⟨*file*$_i$⟩ has an extension, then Oδyssée is looking for the file ⟨*file*$_i$⟩. Otherwise, Oδyssée is looking for the file ⟨*file*$_i$⟩.⟨*extension*⟩ where ⟨*extension*⟩ is the value of the Oδyssée global variable fortran_ext.

load returns an error message if no file of the given name(s) are found in one of the directory listed in the Oδyssée global variable fortran_path. Oδyssée also returns an error message if any file found contains anything else but valid FORTRAN 77 compilation units.

Note that, if there is an include statement in the loaded code, it is necessary to have properly set the Oδyssée global variable include_path.

**remove** ⟨*unit*⟩
Removes the compilation unit of name ⟨*unit*⟩ from the internal library.

remove returns an error message if ⟨*unit*⟩ is not the name of a compilation unit of the internal library.

### 7.2.2 Displaying FORTRAN 77 Code

getlibrary, getunit, getprogram and getdiffprogram are used to view units from the internal library.

Those four commands may all be invoked with an optional ⟨*file*⟩ parameter. If this parameter is present, their output is printed in a file, otherwise it is displayed on the standard input. If the name ⟨*file*⟩ has an extension, then Oδyssée is writing in the file of name ⟨*file*⟩. In the other case, Oδyssée is writing in the file of name ⟨*file*⟩.⟨*extension*⟩ where ⟨*extension*⟩ is the value of the Oδyssée global variable fortran_ext. getlibrary, getunit, getprogram and getdiffprogram are writing FORTRAN 77 files in the directory defined by the Oδyssée global variable fortran_dir. They do not produce any warning message if the file of name ⟨*file*⟩ or of name ⟨*file*⟩.⟨*extension*⟩ already exists in this directory.

The format of the output of these four commands may be tuned by resetting Oδyssée global variables like all_uppercase, do_indentation, if_indentation, continuation_char or blank_line_after_statement, see section 7.8 for further details.

**getlibrary** [⟨*file*⟩]
Displays all the compilation units from the internal library.

```
ODYTOP> load exemple
ODYTOP> getlibrary
      SUBROUTINE head (i1, i2, o)

      common /zz/ zn,zd
      double precision i1,i2,o,zn,zd

      CALL sub0(i1, 12)
      o = zn/zd
      ...
```

**getprogram** ⟨*unit*⟩ [⟨*file*⟩]
Displays the set of the compilation units corresponding to the program whose head unit is the compilation unit of name ⟨*unit*⟩ (see figure 7.1 page 45).

Depending on the computation performed by O∂*y*ssée during the session, getprogram may add comments to the original code of the units giving the informations stored in its information base (dummys, input and output variables).

getprogram returns an error message if ⟨*unit*⟩ is not the name of a compilation unit of the internal library.

**getdiffprogram** ⟨*unit*⟩ [⟨*file*⟩]
Displays the set of the compilation units corresponding to the program whose head unit is the compilation unit of name ⟨*unit*⟩, provided that this unit is the head unit of a program generated by a diff command (see figure 7.2 page 46).

Even if some initial units – not generated by a diff command – belong to the program, getdiffprogram only displays the units generated by the diff command. The code of these units includes some comments providing informations stored in the active information base of O∂*y*ssée.

getdiffprogram returns an error message if the name ⟨*unit*⟩ is not the name of a compilation unit of the internal library generated by a diff command.

```
ODYTOP> getprogram sub2
COD Odyssee  1.7


COD Program : sub2


COD Compilation unit : sub2

      SUBROUTINE sub2 (x, y)

      DOUBLE PRECISION x, y

      y = f(x)-g(x)
      y = sqrt(y)
      RETURN
      END


COD Compilation unit : f

      FUNCTION f (t)

      DOUBLE PRECISION t
      DOUBLE PRECISION f

      f = t*t
      f = exp(f)
      RETURN
      END


COD Compilation unit : g

      FUNCTION g (t)

      DOUBLE PRECISION t
      DOUBLE PRECISION g

      g = 1
      IF (t.NE.0.d0) THEN
        g = sin(t)/t
      END IF
      RETURN
      END
```

Figure 7.1: **getprogram sub2**

```
ODYTOP> diff -tl -h g -vars t
ODYTOP> getdiffprogram gtl
COD Odyssee  1.7

COD Program : gtl



COD Compilation unit : gtl
COD Derivative of unit : g
COD Dummys:  t g
COD Active IN   dummys:  t
COD Active OUT  dummys:  g
COD Dependencies between OUT and IN:
COD g <--   t


      SUBROUTINE gtl (t, g, tttl, gttl)

      IMPLICIT NONE
      INCLUDE 'odyparam.inc'
      DOUBLE PRECISION sd01sttl
      DOUBLE PRECISION tttl
      DOUBLE PRECISION gttl
      DOUBLE PRECISION sd01s
      DOUBLE PRECISION t
      DOUBLE PRECISION g
      DOUBLE PRECISION zn
      DOUBLE PRECISION zd
      COMMON /zz/ zn, zd

      gttl = 0
      g = 1
      IF (t.NE.0.d0) THEN
        sd01sttl = tttl*cos(t)
        sd01s = sin(t)
        gttl = (sd01sttl*t-sd01s*tttl)/t**2
        g = sd01s/t
      END IF
      RETURN
      END
```

Figure 7.2: **getdiffprogram gtl**

**getunit** ⟨*unit*⟩ [⟨*file*⟩]
Displays the compilation unit of name ⟨*unit*⟩.

getunit returns an error message if ⟨*unit*⟩ is not the name of a compilation unit in the internal library.

```
ODYTOP> getunit sub2
      SUBROUTINE sub2 (x, y)

      DOUBLE PRECISION x, y

      y = f(x)-g(x)
      y = sqrt(y)
      RETURN
      END
```

### 7.2.3 Displaying Information about one unit

printunit, getsymbols, getcommons are used to view internal information of a specific unit.

Those three commands may all be invoked with an optional ⟨*file*⟩ parameter. If this parameter is present, their output is printed in a file, otherwise it is displayed on the standard input. If the name ⟨*file*⟩ has an extension, then Oδyssée is writing in the file of name ⟨*file*⟩. In the other case, Oδyssée is writing in the file of name ⟨*file*⟩.⟨*extension*⟩ where ⟨*extension*⟩ is the value of the Oδyssée global variable odyssee_ext. printunit, getsymbols, getcommons are writing Oδyssée files in the directory defined by the Oδyssée global variable odyssee_dir. They do not produce any warning message if the file of name ⟨*file*⟩ or of name ⟨*file*⟩.⟨*extension*⟩ already exists in this directory.

**printunit** ⟨*unit*⟩ [⟨*file*⟩]
Displays the compilation unit on the standard output in an extended format with statements numbers.

printunit returns an error message if ⟨*unit*⟩ is not the name of a compilation unit in the internal library.

```
ODYTOP> printunit sub2
[ Top ]      SUBROUTINE sub2 (x, y)

[ Decl ]      DOUBLE PRECISION x, y, f, g

[ Stat 1 ]      y = f(x)-g(x)
[ Stat 2 ]      y = sqrt(y)
[ Stat 3 ]      RETURN
[ Bottom ]      END
```

**getsymbols** ⟨*unit*⟩ [⟨*file*⟩]
Displays the definition of externals, dummy arguments and local variables of the unit ⟨*unit*⟩.

getsymbols returns an error message if ⟨*unit*⟩ is not the name of a compilation unit in the internal library.

```
ODYTOP> getsymbols sub2
C All external
EXTERNAL f
REAL f
EXTERNAL g
REAL g
C All dummys
DOUBLE PRECISION x
DOUBLE PRECISION y
C All locals
```

**getcommons** ⟨*unit*⟩ [⟨*file*⟩]
Displays the definition of global variables and common blocks of the unit ⟨*unit*⟩. In this version those definitions are global to all the units. What is printed then is the the table of all the units of the library.

getcommons returns an error message if ⟨*unit*⟩ is not the name of a compilation unit in the internal library.

```
ODYTOP> getcommons sub0
COMMON /zz/ zn,zd
DOUBLE PRECISION zd
DOUBLE PRECISION zn
```

### 7.2.4 Displaying Information about the library

listunits, getgraph, and libgraph are used to visualize informations on the contents of the internal library.

The three first commands may all be invoked with an optional ⟨*file*⟩ parameter. If this parameter is present, their output is printed in a file, otherwise it is displayed on the standard output. If the name ⟨*file*⟩ has an extension, then O$\partial$*yssée* is writing in the file of name ⟨*file*⟩. In the other case, O$\partial$*yssée* is writing the file of name ⟨*file*⟩.⟨*extension*⟩ where ⟨*extension*⟩ is the value of the O$\partial$*yssée* global variable odyssee_ext. listunits, getgraph, and libgraph are writing O$\partial$*yssée* files in the directory defined by the O$\partial$*yssée* global variable odyssee_dir. They do not produce any warning message if the file of name ⟨*file*⟩ or of name ⟨*file*⟩.⟨*extension*⟩ already exists in this directory.

**listunits** [⟨*file*⟩]
Displays the list of the compilation units of the internal library: for each compilation unit, its name and its type (Function or Subroutine) are listed.

```
ODYTOP> load exemple
ODYTOP> listunits
Function f
Function g
Subroutine main
Subroutine sub0
Subroutine sub1
Subroutine sub2
```

**libgraph** [⟨*file*⟩]
Displays a representation of the call graph of all the compilation units in the internal library.

```
ODYTOP> libgraph

main +- sub0 +- sub1 +- f
                     +- g
             +- sub2 +- f
                     +- g
```

**getgraph** ⟨*unit*⟩ [⟨*file*⟩]
Displays the sub-graph of the call graph whose head unit is the unit of name ⟨*unit*⟩.

getgraph returns an error message if the name ⟨*unit*⟩ is not the name of a compilation unit of the internal library.

```
ODYTOP> getgraph sub1

sub1 +- f
     +- g
```

## 7.3 The preparation of the code Command

preprocess applies the transformations listed in section 3.1 on the original routines. preprocess produces new compilation units and stores them within the internal library. This computation is automatically performed before any differentiation.

**preprocess** ⟨*unit*⟩
preprocess ⟨*unit*⟩ applies the preprocessing transformations to all the compilation units within the program whose head unit is the compilation unit of name ⟨*unit*⟩

preprocess returns an error message if the name ⟨*unit*⟩ is not the name of a compilation unit of the internal library.

## 7.4 The Information Base Related Commands

### 7.4.1 Setting entries into the Information Base

setdummys, setglobals, setinout, setabstract allow the user to provide O$\partial$yssée with information on the variables of the compilation units. They are particularly useful to provide information on variables of black-box subroutines whose code is not available. They can also be used to overwrite information computed by O$\partial$yssée.

These four commands are not supposed to be used at the interpreter level: they are normally used inside information base files.

**setdummys** ⟨*unit*⟩ ⟨*var_list*⟩
Sets the list of the arguments of the unit of name ⟨*unit*⟩ to the list of variables ⟨*var_list*⟩.

**setglobals** ⟨*unit*⟩ ⟨*var_list*⟩
Sets the list of the global variables of the unit of name ⟨*unit*⟩ to the list of variables ⟨*var_list*⟩.

**setinout** ⟨*unit*⟩ ⟨*var_list_1*⟩ ⟨*var_list_2*⟩ ⟨*var_list_3*⟩ ⟨*var_list_4*⟩
Sets the lists of the input and output dummies and global variables of the unit of name ⟨*unit*⟩.

The list of input dummies is set to the list of variables ⟨*var_list_1*⟩, the list of output dummies to the list ⟨*var_list_2*⟩, the list of input global to the list ⟨*var_list_3*⟩, and the list of output global to ⟨*var_list_4*⟩.

As a side effect, setinout sets the dependencies between variables to their default value: each output variable depends on all the input variables.

**setabstract** ⟨*unit*⟩ ⟨*var_list_list*⟩
Sets the lists of the dependencies between the input and the output variables of the compilation unit of name ⟨*unit*⟩.

### 7.4.2 Computing the Information Base

computeibasis makes O$\partial$yssée analyze the code to find the dependencies between variables. This computation is automatically performed before any differentiation.

getibasis makes O$\partial$yssée store the information base within an information base file for further differentiation of the same original code. This file can be read by O$\partial$yssée using the *loadibasis* command.

**computeibasis** ⟨*unit*⟩
Computes the information base of the program whose head unit has the name ⟨*unit*⟩.

computeibasis returns an error message if the name ⟨*unit*⟩ is not the name of a compilation unit of the internal library.

**getibasis** ⟨*unit*⟩ [⟨*file*⟩]
Prints the information base of the program whose head unit is the unit ⟨*unit*⟩.

If the optional argument ⟨*file*⟩ is present, their output is printed in a file, otherwise it is displayed on the standard input. If the name ⟨*file*⟩ has an extension, then Oδyssée is writing in the file of name ⟨*file*⟩. In the other case, Oδyssée is writing the file of name ⟨*file*⟩.⟨*extension*⟩ where ⟨*extension*⟩ is the value of the Oδyssée global variable ibasis_ext. It write an ibasis file in the directory defined by the Oδyssée global variable ibasis_dir. It does not produce any warning message if the file of name ⟨*file*⟩ or of name ⟨*file*⟩.⟨*extension*⟩ already exists in this directory.

getibasis returns an error message if the name ⟨*unit*⟩ is not the name of a compilation unit of the internal library.

```
ODYTOP> getibasis sub1
setdummys sub1 (x y)
setglobals sub1 ()
setinout sub1 (x) (y) () ()
setabstract sub1 (y (x))

setdummys f (t f)
setglobals f ()
setinout f (t) (f) () ()
setabstract f (f (t))

setdummys g (t g)
setglobals g ()
setinout g (g t) (g) () ()
setabstract g (g (g t))
```

**loadibasis** ⟨$ibasis_1$⟩ [⟨$ibasis_2$⟩ ... ⟨$ibasis_N$⟩]

Attempts to execute the specified information base file(s).

If the name ⟨$ibasis_i$⟩ has an extension, then Oδyssée is looking for the file ⟨$ibasis_i$⟩. Otherwise, Oδyssée is looking for the file ⟨$ibasis_i$⟩.⟨*extension*⟩ where ⟨*extension*⟩ is the value of the Oδyssée global variable ibasis_ext.

loadibasis returns an error message if no file of the given name(s) is found in one of the directory listed in the Oδyssée global variable ibasis_path. Oδyssée also returns an error message if any file found contains anything else but a sequence of valid command lines of this language.

If the execution of an ibasis file produces an error then the command lines of the next files are not executed.

### 7.4.3 Displaying Information from the Information base

getinout, getabstract, getdependency are used to visualize informations stored in the information base.

Those two commands may both be invoked with an optional ⟨*file*⟩ parameter. If this parameter is present, their output is printed in a file, otherwise it is displayed on the standard input. If the name ⟨*file*⟩ has an extension, then O∂yssée is writing in the file of name ⟨*file*⟩. In the other case, O∂yssée is writing the file of name ⟨*file*⟩.⟨*extension*⟩ where ⟨*extension*⟩ is the value of the O∂yssée global variable odyssee_ext. getinout, and getabstract are writing O∂yssée files in the directory defined by the O∂yssée global variable odyssee_dir. getibasis writes an information base file They do not produce any warning message if the file of name ⟨*file*⟩ or of name ⟨*file*⟩.⟨*extension*⟩ already exists in this directory.

**getinout** ⟨*unit*⟩ [⟨*file*⟩]
Displays the input and the output variables of the compilation units belonging to the program whose head unit is the unit ⟨*unit*⟩.

getinout returns an error message if the name ⟨*unit*⟩ is not the name of a compilation unit of the internal library.

```
ODYTOP> getinout sub0
Compilation unit : sub0
IN   dummys: v u
IN  globals: zd zn
OUT  dummys: v u
OUT globals: zd zn

Compilation unit : sub1
IN   dummys: y x
OUT  dummys: y x

Compilation unit : f
IN   dummys: f t
OUT  dummys: f t

Compilation unit : g
IN   dummys: g t
OUT  dummys: g t

Compilation unit : sub2
IN   dummys: y x
OUT  dummys: y x
```

**getabstract** ⟨*unit*⟩ [⟨*file*⟩]
Displays the dependencies between the input and the output variables of the compilation units belonging to the program whose head unit is the unit ⟨*unit*⟩.

getabstract returns an error message if the name ⟨*unit*⟩ is not the name of a compilation unit of the internal library.

```
ODYTOP> getabstract sub0
Compilation unit : sub0
COD Dependencies between OUT and IN:
u <--  zd zn v u
v <--  zd zn v u
zn <--  zd zn v u
zd <--  zd zn v u

Compilation unit : sub1
x <--  y x
y <--  y x

Compilation unit : f
t <--  f t
f <--  f t

Compilation unit : g
t <--  g t
g <--  g t

Compilation unit : sub2
x <--  y x
y <--  y x
```

**getdependency** ⟨*unit*⟩ [⟨*file*⟩]
Displays the table of dependency table of the compilation unit ⟨*unit*⟩. In this version, this table is not stored any more, so it will return nothing.

getinout returns an error message if the name ⟨*unit*⟩ is not the name of a compilation unit of the internal library.

## 7.5 The Differentiation Related Commands

### 7.5.1 The diff Command

The command diff is used to differentiate a program stored in the internal library. diff produces new compilation units and stores them in the internal library. The syntax of diff is the following:

**diff** ⟨*arguments_expression*⟩
⟨*arguments_expression*⟩ is made up of several necessary and optional arguments given as:

-⟨*argument_name*⟩ [⟨*argument_value*⟩]

To run properly, diff requires at least three arguments:

- the name of the head unit to differentiate,
- the list of the active input variables,
- the differentiation method.

But the user can also specify the way the derivative is printed using optional arguments.

The table below gives the list of the different options corresponding to the mode of differentiation.

| | | |
|---|---|---|
| -tangent | -tl | *diff -tangent* or *diff -tl*<br>Sets the differentiation method to direct mode.<br>The generated code will compute the *tangent code* of the function (see command line page 19 and example of generated code in figure 3.1). |
| -cotangent | -cl | *diff -cotangent* or *diff -cl*<br>Sets the differentiation method to reverse mode. The generated code will compute the *cotangent code* of the function. By default it uses the "*syntax reversal*" algorithm and does not handle codes with `goto, exit` (see command line page 21 and example of generated code in figure 3.2). |
| -cotangent -goto | -cl -goto | *diff -cl -goto*<br>Changes the default method to the "*flow reversal*" algorithm (see command line page 23 and example of generated code in figure 4.3). |
| -cotangent -optloops | -cl -opt | *diff -cl -optloops* ⟨*unit*⟩ or *diff -cl -opt* ⟨*unit*⟩<br>Tells O∂yssée to use the *checkpointing* facility on the loop around the call of ⟨*unit*⟩ (see command line page 26 and example of generated code in figure 4.2). This method can be used only if the files opt_rev.f and opt_utils.f have been load in the internal library. These two files are included in the sub-directory fortran of the O∂yssée distribution. |

The table below gives the arguments to chose the head unit and active input variables.

| | | |
|---|---|---|
| -head | -h | -head ⟨*unit*⟩ or -h ⟨*unit*⟩<br>Introduces the head unit of the program to differentiate. diff will return an error message if the name ⟨*unit*⟩ is not the name of a compilation unit of the internal library. |
| -variables | -vars | -variables ⟨*var_list*⟩ or -vars ⟨*var_list*⟩<br>Introduces the list of the active variables for differentiation |

The table below gives the list of optional arguments of diff that defines the output of the diff command, their syntax and their actions. By default, the FORTRAN 77 code generated by diff is stored in the internal library but is neither displayed nor saved in a file.

| | | |
|---|---|---|
| -nolib | | *diff -nolib*<br>Changes the default output. The FORTRAN 77 code generated by diff is not stored and must then be printed using -output or -split. |
| -output | -o | *diff -output* ⟨*file*⟩ or *diff -o* ⟨*file*⟩<br>Giving std as file name tells Oâyssée to display the code on the standard output. Otherwise the code is printed in ⟨*file*⟩ using the same rule than other commands displaying FORTRAN 77 code (see section 7.2.2). |
| -split | | *diff -split*<br>Tells Oâyssée to print each unit of the differentiated program to a file which name is the name of the unit using the same rule than other commands displaying FORTRAN 77 code (see section 7.2.2) for the path and the extension. |

### 7.5.2 Other Commands

A side effect of the evaluation of the diff -head ⟨*unit*⟩ -vars ⟨*var_list*⟩ command line is to compute the active dependencies i.e. to propagate the active input variables ⟨*var_list*⟩ through the compilation units of the program which head unit has name ⟨*unit*⟩. These dependencies are stored in the active information basis attached to the diff computation. getactiveinout can be used to visualize the information of the last computed active information basis.

**getactiveinout** ⟨*unit*⟩ [⟨*file*⟩]
Displays the dependencies between the active input variables and the active output variables of each unit of the program which head unit has name ⟨*unit*⟩.

If the ⟨*file*⟩ parameter is given, the output of getactiveinout is redirected to a file, it is displayed on the standard output otherwise.

If the name ⟨*file*⟩ has an extension, then O∂yssée is writing in the file of name ⟨*file*⟩. In the other case, O∂yssée is writing the file of name ⟨*file*⟩.⟨*extension*⟩ where ⟨*extension*⟩ is the value of the O∂yssée global variable odyssee_ext. When writing in a file, getactiveinout are writing in the directory defined by the O∂yssée global variable odyssee_dir. It does not produce any warning message if the file of name ⟨*file*⟩ or of name ⟨*file*⟩.⟨*extension*⟩ already exists in this directory.

## 7.6 The Units Related Commands

**slice** ⟨*unit*⟩ ⟨*var_list*⟩
Generates an unit equivalent to the unit of name ⟨*unit*⟩ where the code is restricted to the computation of the variables of the list ⟨*var_list*⟩.

**makeblock** ⟨*unit*⟩ ⟨*main*⟩ ⟨*sub*⟩ ⟨*number_list*⟩
Builds the subroutine of name ⟨*sub*⟩ with the statements of the routine of name ⟨*unit*⟩ whose index are in the list ⟨*number_list*⟩. Create the routine of name ⟨*main*⟩ which is equivalent to the routine of name ⟨*unit*⟩ but contains a call to the subroutine ⟨*sub*⟩.

```
ODYTOP> printunit sub0
[ Top ]      SUBROUTINE sub0 (u, v)

[ Decl ]      COMMON /zz/ zd, zn
[ Decl ]      DOUBLE PRECISION u, v, z1, z2, zn, zd

[ Stat 1 ]      CALL sub1(u, z1)
[ Stat 2 ]      CALL sub2(v, z2)
[ Stat 3 ]      zn = z1-z2
[ Stat 4 ]      zd = 1+z1+z2
[ Stat 5 ]      RETURN
[ Bottom ]      END

ODYTOP> makeblock sub0 sub0s sub3 (3 4)

ODYTOP> getunit sub0s
      SUBROUTINE sub0s (u, v)

      EXTERNAL sub3
      COMMON /zz/ zd, zn
      DOUBLE PRECISION u, v, z1, z2, zn, zd

      CALL sub1(u, z1)
```

```
      CALL sub2(v, z2)
      CALL sub3(z2, z1)
      RETURN
      END

ODYTOP> getunit sub3
      SUBROUTINE sub3 (z2, z1)

      COMMON /zz/ zd, zn
      DOUBLE PRECISION z1, z2, zn, zd

      zn = z1-z2
      zd = 1+z1+z2
      RETURN
      END
```

## 7.7 The System Related Commands

### 7.7.1 Loading Batch Files

**loadbatch** $\langle batch_1 \rangle$ [$\langle batch_2 \rangle$ ... $\langle batch_N \rangle$]
Attempts to execute the specified batch file(s).

If the name $\langle batch_i \rangle$ has an extension, then Oδyssée is looking for the file $\langle batch_i \rangle$. Otherwise, Oδyssée is looking for the file $\langle batch_i \rangle . \langle extension \rangle$ where $\langle extension \rangle$ is the value of the Oδyssée global variable batch_ext.

loadbatch returns an error message if no file of the given name(s) is found in one of the directory listed in the Oδyssée global variable batch_path. Oδyssée also returns an error message if any file found contains anything else but a sequence of valid command lines of this language.

If the execution of a batch file produces an error then the command lines of the next files are not executed.

**loadodyseerc**
Attempts to execute the special init batch file HOME/.odysseerc

### 7.7.2 Miscellaneous Commands

**help** [$\langle command \rangle$]
help without argument displays a list of all the commands of the Oδyssée language.

help followed by a string which is the name of a command of the Oδyssée language displays a short description of this command including its syntax.

help followed by anything else produces an error message.

**shell** ⟨*string*⟩
Sends ⟨*string*⟩ to the shell.

**version**
Displays the current version number of O∂yssée.

**copyright**
Displays the copyright banner of O∂yssée.

**exit**
Terminates O∂yssée without saving anything.

**quit**
Terminates O∂yssée without saving anything.

## 7.8 Configuration of the System

In this section, all the O∂yssée global variables are described. Initialized to their default values when O∂yssée starts, these variables allow the user to change the configuration of the system.

### 7.8.1 Command for the configuration of the System

**getvar** ⟨*var*⟩
Displays the value of the O∂yssée global variable ⟨*var*⟩ (see 7.8).

**setvar** ⟨*var*⟩ ⟨*value*⟩
Sets the value of the variable ⟨*var*⟩ to ⟨*value*⟩.

setvar returns an error message if the type of ⟨*value*⟩ is not compatible with the predefined type of ⟨*var*⟩ (see 7.8).

### 7.8.2 Option of code canonicalization

minimal_split is the only global variable used for the configuration of the canonicalization method. It is a boolean variable whose default value is true. When false, O∂yssée splits all the algebraic expression in the input code into elementary operations, otherwise, the constants and the arguments of the functions are isolated in local variables.

### 7.8.3 Format of Generated FORTRAN 77 Code

| Variable | Type | Default | Comment |
|---|---|---|---|
| all_uppercase | boolean | true | When false, only the FORTRAN 77 keywords of the generated code are displayed with uppercase letters, otherwise, the whole code is written with uppercase letters. |
| do_indentation | integer | 3 | Tunes the indentation of the body of a do loop in the generated code. Unit is a white space. |
| if_indentation | integer | 2 | Tunes the indentation of the body of a if statement in the generated code. Unit is a white space. |
| continuation_char | string | : | Defines the continuation char. |
| blank_line_after_stat | boolean | false | When true, an empty line is inserted after any statement in the FORTRAN 77 generated code. |
| parenthesis_level | integer | 0 | Tunes the level of parenthesis in the generated instructions. Default correspond to a minimal level of parenthesis. |

### 7.8.4 Input and Output through Files

| Variable | Type | Default | Comment |
|---|---|---|---|
| fortran_ext | string | f | Extension of the FORTRAN 77 files (read and writen) |
| fortran_path | list of strings | (.) | List of the paths used to read FORTRAN 77 files. |
| fortran_dir | string | . | Name of the directory where O$\partial$yssée writes the FORTRAN 77 files. |
| include_ext | string | inc | Extension of the include files (read and writen) |
| include_path | list of strings | (.) | List of the paths used to read include files of the FORTRAN 77 code. |
| include_dir | string | . | Name of the directory where O$\partial$yssée writes the include files. |
| batch_ext | string | batch | Extension of the batch files (read and writen) |
| batch_path | list of strings | (.) | List of the paths used to read batch files. |
| batch_dir | string | . | Name of the directory where O$\partial$yssée writes the batch files. |
| ibasis_ext | string | bi | Extension of the information base files (read and writen) |
| ibasis_path | list of strings | (.) | List of the paths used to read information base files. |
| ibasis_dir | string | . | Name of the directory where O$\partial$yssée writes the information base files. |
| odyssee_ext | string | od | Extension of the file to be written which may not be read again by O$\partial$yssée. |
| odyssee_dir | string | (.) | Name of the directory where O$\partial$yssée writes the files which are not supposed to be read again. |
| home_dir | string | HOME | Home dir of the user (O$\partial$yssée point of view). |

# Chapter 8

# The O$\partial y$ssée Graphical User Interface

The graphical user interface has been designed with the help of the TK toolkit. Therefore it has a MOTIF look-and-feel with pop-up windows, menus and sub-menus, highlighting of pre-selected and selected items, radio buttons, scrollable windows ...see figure 8.1.

The main window of the O$\partial y$ssée graphical user interface consists in three different parts :

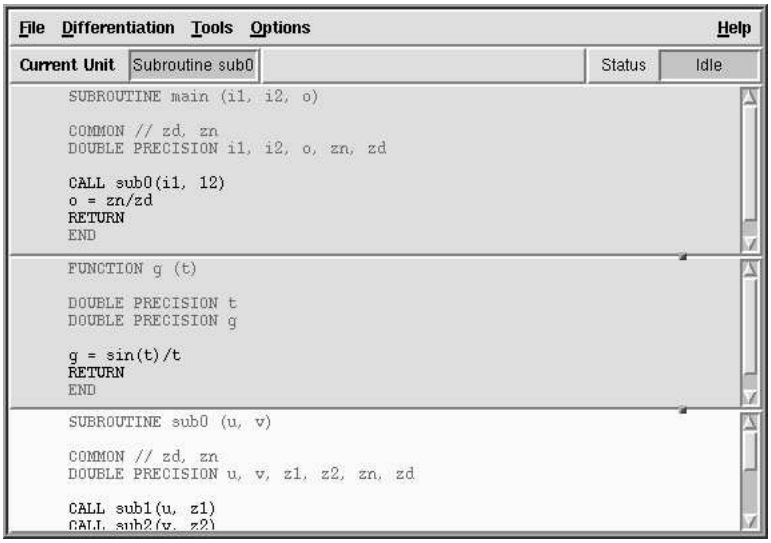
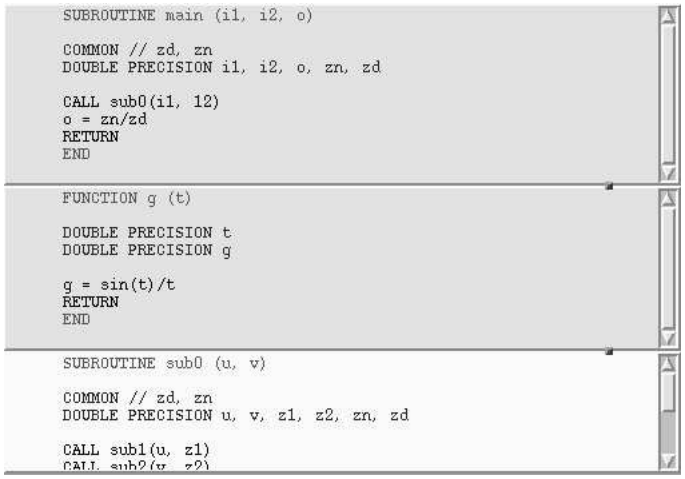



Figure 8.1: the O$\partial y$ssée graphical user interface window

- the menubar with four active menus used for sending commands to O$\partial$yssée, and a Help menu,


- the status bar with an active button to select the *current unit* in the internal library, and two information windows displaying the name of the current unit and the status of the engine of O$\partial$yssée.


- the display frame, which can be divided in several views, used for the visualization of the FORTRAN 77 units of the internal library



```
      SUBROUTINE main (i1, i2, o)

      COMMON // zd, zn
      DOUBLE PRECISION i1, i2, o, zn, zd

      CALL sub0(i1, 12)
      o = zn/zd
      RETURN
      END
```

```
      FUNCTION g (t)

      DOUBLE PRECISION t
      DOUBLE PRECISION g

      g = sin(t)/t
      RETURN
      END
```

```
      SUBROUTINE sub0 (u, v)

      COMMON // zd, zn
      DOUBLE PRECISION u, v, z1, z2, zn, zd

      CALL sub1(u, z1)
      CALL sub2(v, z2)
```

In the current version of O$\partial$yssée, the Help menu is restricted to one item 

which provides the informations on the version of O$\partial$yssée in a pop-up window as shown below (Click on Ok to dismiss it).


Figure 8.2: the File menu

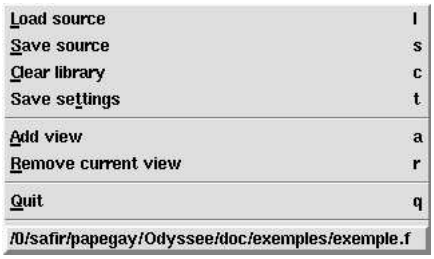



Figure 8.3: the extended File menu

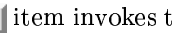
## 8.1 The File Menu

There is seven items in the File menu when beginning a new session of O$\partial$yssée (see figure 8.2).

The Save item stays grayed and the Save feature is disabled as long as the internal library is empty. When FORTRAN 77 files are loaded in the internal library of O$\partial$yssée, the File menu is extended by items labeled with the names of these files (see figure 8.3).

Selecting one of those items is a shortcut to reloading the corresponding file in the internal library of O$\partial$yssée.

### 8.1.1 The Load source Button

The 

item invokes the load command of the language of O$\partial$yssée to load in the internal library the compilation units of selected file(s). A dialog window like in figure 8.4 appears on the screen for the selection of the file(s) to load.

In the display area of the dialog-window are listed the files of the directory whose name appears in the Pathname field and matches the value of the Selection pattern field.

Files are selected by clicking on them, then the name(s) of selected file(s) are grayed in the display area and appear(s) in the Filename field.

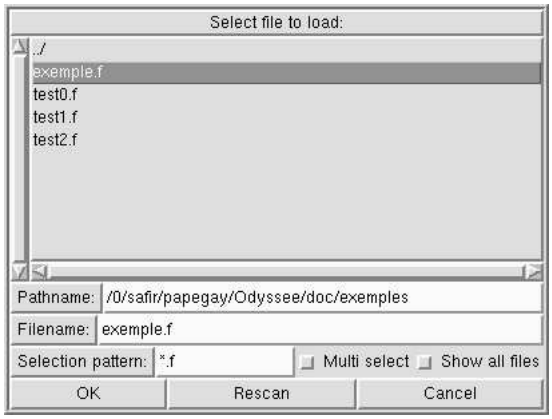



Figure 8.4: the Load source dialog window

When the selected file is a directory, its name is appended to the pathname by double-clicking on it and then the files corresponding to the selection pattern are listed in the display area.

Clicking on the Selection pattern button offers a selection of possible extensions.

Pathname, Filename and Selection pattern fields can also be edited in the corresponding frames.

Multiple selection is allowed when the Multi select check-button is activated. Double-clicking on a name will append it to the Filename list.

Activation of the Show all files check-button causes O$\partial$yssée to list all the files of the Pathname directory – even the files that start with a dot – in the display area.

Clicking on the Cancel button interrupts the loading process and causes O$\partial$yssée to return to its previous state.

The Rescan button is used for updating the list of files selectable.

Once the selection has been done, clicking on the Ok button invokes the load command with the selected file(s) as argument(s).

An error window is popped up if no file of the selected name is found in the pathname directory or if any found file is a syntactically incorrect FORTRAN 77 file.

### 8.1.2 The Save source Button

When enabled, the Save source s item invokes the getunit command of the command language of O$\partial$yssée to save the current unit in the selected file. A dialog window like in figure 8.5 appears on the screen for the selection of the file.

In the display area of the dialog window, the files of the directory which name has the value of the Pathname field and matching the value of the Selection pattern field are listed.

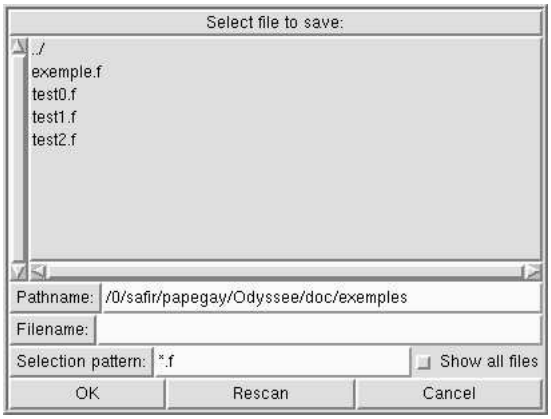



Figure 8.5: the Save source dialog window

A file can be selected by clicking on its name, then the name of the selected file is grayed in the display area and appears in the Filename field.

When the selected file is a directory, its name is appended to the pathname by double-clicking on it and then the files corresponding to the selection pattern are listed in the display area.

Clicking on the Selection pattern button offers a selection of possible extensions.

Pathname, Filename and Selection pattern fields can also be edited in the corresponding frames.

Activation of the Show all files check-button causes O$\partial$yssée to list all the files – even the files that start with a dot – of the pathname directory in the display area.

Clicking on the Cancel button interrupts the saving process and causes O$\partial$yssée to return to its previous status.

The Rescan button is used for updating the list of selectable files.

Once the selection has been done, clicking on the Ok button invokes the save command with the current unit and the selected file as argument.

If the file already exists, it will be overwritten without any warning.

### 8.1.3 Other Buttons

Selecting the Clear library c resets the internal library of O$\partial$yssée to the empty set.

The Save settings t saves the current values of several O$\partial$yssée global variables (governing paths and extension names) into a batch file of name .xodysseerc.batch in the directory where O$\partial$yssée has been started. Note that this file is different from the .odysserc file and is not loaded automatically when starting a new session of O$\partial$yssée.

The views management buttons will be described in the section 8.2 which presents the display area and the features to control this area.

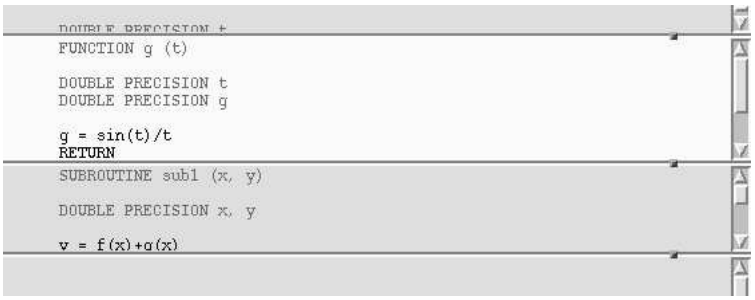
Clicking on Quit q terminates O∂yssée without saving anything but a pop-up window appears for confirmation.

## 8.2 The Display Area

To visualize several compilation units at the same time, the display area can be divided in several views.

The Add view a button adds a new view in the display area. As the size of the display area is controlled by the window manager, the existing views are resized when adding a new view: the size of each view is set to the size of the display area divided by the number of views.

The current view can be selected by clicking on it. All the views, except the *current view*, are grayed. The respective size of the views can be changed by the user by moving the separators lines (using a small drop-and-drag button on the line) as shown below:




The information of the Current Unit of the status bar concerns the current view.

The Remove current view r button removes the current view of the display area. Whatever the size of the remaining views, they are resized: the size of each view is set to the size of the display area divided by the number of views. The view on top of the display area is set as the current view, and the attached compilation unit becomes the current unit.

The left part of the status bar consists in a button labeled Current unit and a window where the name of the current unit is displayed:


By definition, the current unit is the compilation unit displayed in the current view.

Clicking on the Current unit button invokes the listunits command and pulls up a menu like in figure 8.6 with as many (up to ten) items as units in the internal library of O∂yssée. Clicking on one of those items selects the corresponding compilation unit as current unit. If the internal library contents more than ten compilation units then the only ten first units are displayed and a More... item allows to access a new menu with other units.

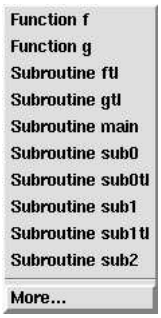
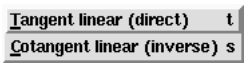



Figure 8.6: a sample of the Current unit button

## 8.3 The Differentiation Menu




The two items of the Differentiation menu correspond to a call to the diff command on the current unit with different options for the method of differentiation, respectively -tl for direct (tangent line computation) and -cl for inverse (cotangent line computation).

The active input variables of the head unit are required prior to differentiating FORTRAN 77 code. For this purpose, invocation of any of these two items causes O*dyssée* to open a new dialog window (see figure 8.7).

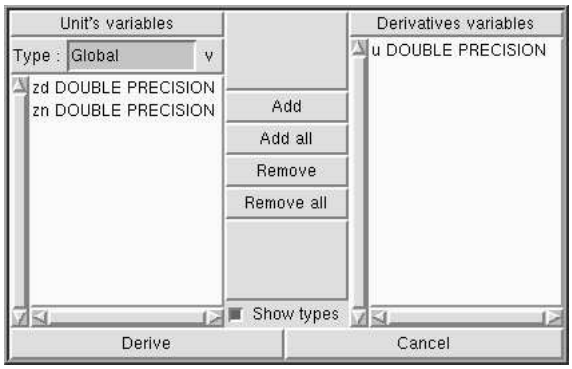



Figure 8.7: the dialog window for Differentiation menu

The active input variables with respect to which O*dyssée* will differentiate the current unit have to be selected by the user.

The right part of the dialog window is the Unit's variables frame. In this frame is displayed the list of the variables of the current unit.

The variables are listed by type and only the variables of types Global, Local or Dummy appearing in the frame Type are displayed – only the Global variables in figure 8.7.

To change the type of the variables to be displayed, click on the v button and select the item of the menu corresponding to the selected type. Note that it is impossible to differentiate with respect to a local variable.

The selected active input variables are listed on the left part of the dialog window: the Derivatives variables frame.

To add a variable or a list of variables displayed in the Unit's variables frame to the Derivatives variables frame, select their names by clicking on them and click on the Add button.

Use Add all button to add all the variables displayed in the Unit's variables frame to the Derivatives variables frame.

To remove a variable displayed in the Derivatives variables frame from the list of the selected active input variables, select its name by clicking on it and click on the Remove button.

Use Remove all button to remove all the variables displayed in the Derivatives variables frame from the list of the selected active input variables.

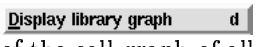
The Show types check-button commands the display of the FORTRAN 77 type of the variables.

Click on the Derive button to send the diff command to the O$\partial$yssée engine, on the Cancel button to abort it.

## 8.4 The Tools Menu


The Tools menu consists of six items but in the current version of O$\partial$yssée, only five of them are implemented. That is why the Optimize unit stays grayed.

### 8.4.1 The Display library graph Button

The 

button invokes the libgraph command and displays a representation of the call graph of all the compilation units in the internal library in a pop-up window as shown in figure 8.8.

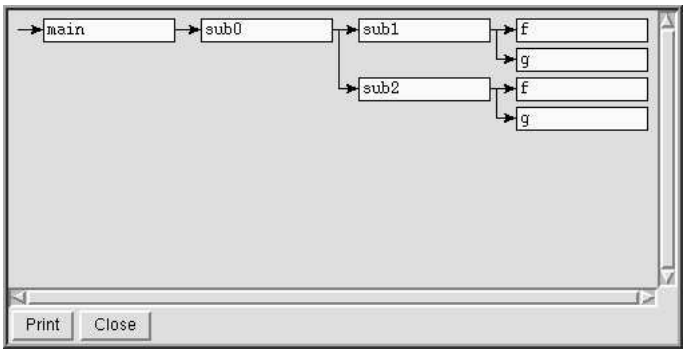



Figure 8.8: a sample of a call graph

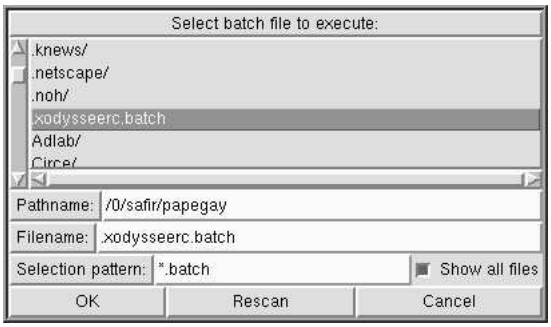



Figure 8.9: the Execute batch file dialog window

### 8.4.2 The Execute batch file Button

The Execute batch file e button is used to execute the sequence of commands of a batch-file into O$\partial$yssée. Selection of the batch file is made through a dialog window like in figure 8.9.

In the display area of the dialog-window are listed the files of the directory whose name has the value of the Pathname field and matching the value of the Selection pattern field.

Files are selected by clicking on them, then the name(s) of selected file(s) are grayed in the display area and appear(s) in the Filename field.

When the selected file is a directory, its name is appended to the pathname by double-clicking on it and then the files corresponding to the selection pattern are listed in the display area.

Clicking on the Selection pattern button offers a selection of possible extensions.

Pathname, Filename and Selection pattern fields can also be edited in the corresponding frames.

Activation of the Show all files check-button causes O$\partial$yssée to list all the files of the pathname directory – even the files that start with a dot – in the display area.

Clicking on the Cancel button interrupts the loading process and causes O$\partial$yssée to return to its previous status.

The Rescan button is used for updating the list of selectable files.

Once the selection has been done, clicking on the Ok button invokes the loadbatch command with the selected file as argument and the command lines of the batch-file are executed by O$\partial$yssée.

An error window is popped up if no file with the selected name is found in the pathname directory or if any files found contain anything else but valid O$\partial$yssée command lines.

### 8.4.3 The Make block Button

Clicking on the 

button is only possible when statements of the current unit have been selected, otherwise the item stays greyed. It invokes the menublock command to build a subroutine with the selected statements of the current unit and a main routine which is equivalent to the current unit and contents a call to the subroutine (see exemple on figure 8.10). Names of the main routine and of the subroutine are prompted in a pop-up window.

### 8.4.4 The Edit current unit Button

Clicking on the 

button opens a window for editing the FORTRAN 77 code of the current unit (see figure 8.11).

The editing facilities are the features of a very simplified emacs-like editor accepting the control characters described in the table below where "^" stands for "control".

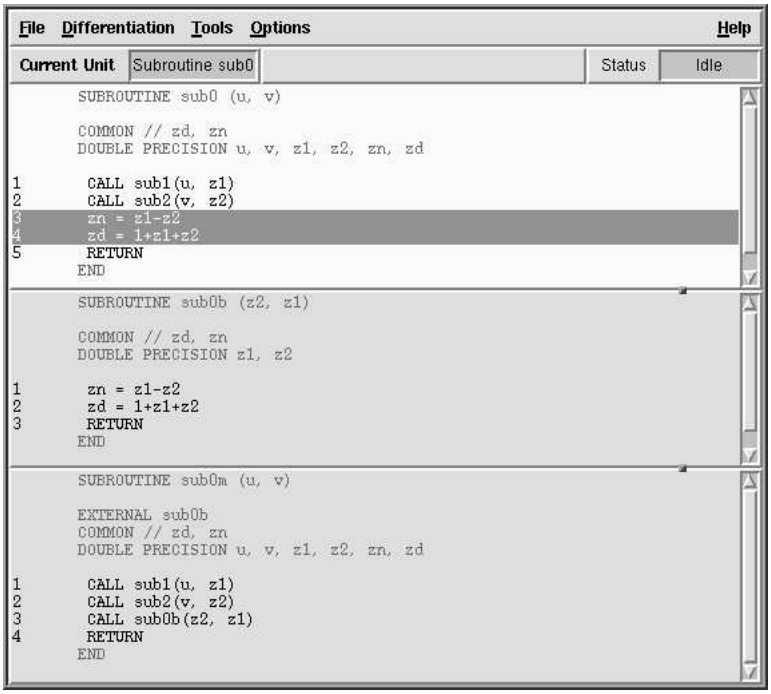



Figure 8.10: An exemple of use of Make block button

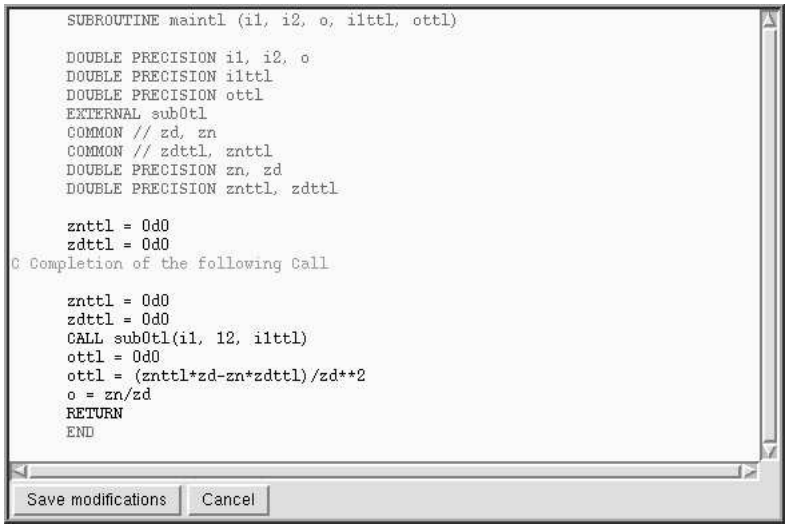



Figure 8.11: the editing window for FORTRAN 77 code

| characters | action |
|---|---|
| ^f | forward character |
| ^b | backward character |
| ^a | beginning of line |
| ^e | end of line |
| ^p | previous line |
| ^n | next line |
| ^d | delete character |
| ^h | delete backward |
| ^t | transpose characters |
| ^k | kill line |
| ^o | open line |

The effect of the Save modifications button is to replace the original unit by the result of the edition inside the internal library of O$\partial$yssée: if the name of the routine has been modified during edition, then the result of the edition is saved under the new name as a new unit of the internal library, otherwise, the original routine is overwritten.

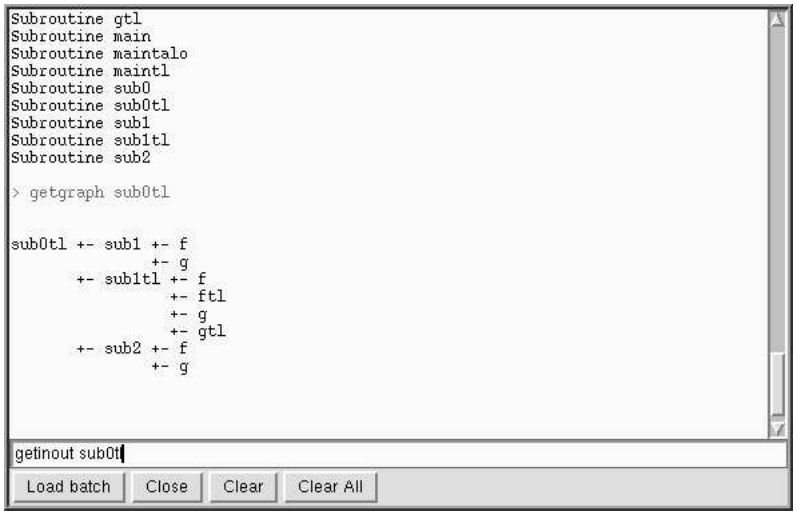



Figure 8.12: the interpreter window

If the result of the edition is not a valid FORTRAN 77 compilation unit, then O*δyssée* pops up an error window.

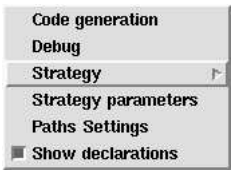
Clicking on the Cancel button aborts the edition process without changing anything.

### 8.4.5 The Send command to Odyssee Button

The Send command to Odyssee button allows to use the command language of O*δyssée* without exiting the graphical user interface. Commands are entered and results are displayed in an O*δyssée* command window like shown in figure 8.12.

In the upper part of the window are displayed the result of the O*δyssée* commands in the same format than they would be on an interpreter window. The lower part of the window is used for editing a line of commands The following control characters are recognized for edition (`^f`: forward character, `^b`: backward character, `^a`: beginning of line, `^e`: end of line).

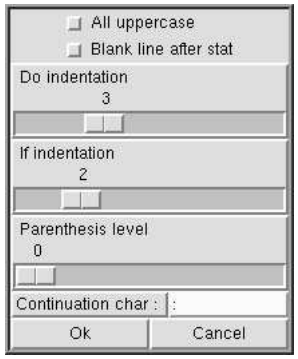
## 8.5 The Options Menu




The six buttons of the Options are used for setting global variables of O$\partial$yssée.

### 8.5.1 The Code generation Button

The Code generation button is used to control the format of the FORTRAN 77 code printed by O$\partial$yssée. It pops up the window below:




When the All uppercase check-button is not on, only the FORTRAN 77 keywords of the printed code are written with uppercase letters, otherwise, the whole code is written with uppercase letters included the names of the variables.

When the Blank line after stat check button is checked, an empty line is inserted after any statement in the FORTRAN 77 printed code.

The Do indentation slide button tunes the indentation of the body of a do loop in the printed code. The unit is a white space.

The If indentation slide button tunes the indentation of the body of a if statement in the printed code. The unit is a white space.

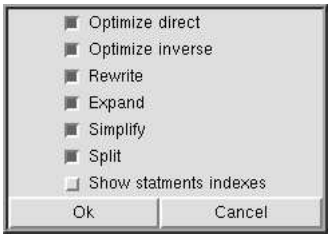
The Parenthesis level slide button tunes the level of parenthesis in the printed instructions. Zero corresponds to a minimal level of parenthesis

The Continuation char input field allows to enter the FORTRAN 77 continuation character used when breaking lines of code.

### 8.5.2 The Debug Button

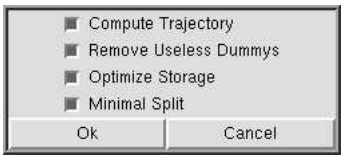
The Debug button is for developers purpose only. It allows to suppress (or to add) parts of the differentiation process executed by the diff command.

The Debug dialog box containing seven check-buttons is reproduced below:




### 8.5.3 The Strategy parameters Button

The Strategy parameters button is used for controlling the strategy used by the differentiation algorithm of Oϑyssée. Four strategy options can be chosen by checking buttons of the dialog window:




In the current version, only the strategy associated to the Minimal split button is implemented.

When the button is not checked, Oϑyssée splits all the algebraic expression in the input code into elementary operations, otherwise, the constants and the arguments of the functions are isolated in local variables.

### 8.5.4 The Path settings Button

The Paths Settings button is used to set the path and the extensions of the names of files for input or output. It pops up the window below:


The several input fields allow the user to redefine the paths and the extensions of files read or written by Oϑyssée (FORTRAN 77 units, include FORTRAN 77 files, batch files, information base files).

## 8.6 Other Items

The Strategy button is not yet implemented.

The Show declarations button is a check button which control the display of the FORTRAN 77 compilation units. The declarations of variables inside the FORTRAN 77 routines are shown only when this button is checked.

# Bibliography

[1] M. Berz, C.H. Bischof, G.F. Corliss, and A. Griewank, editors. *Computational Differentiation: Applications, Techniques, and Tools*. SIAM, 1996.

[2] I. Charpentier and M. Ghemires. Génération automatique de codes adjoints : Stratégies d'utilisation pour le logiciel odyssée, application au code météorologique meso-nh. Rapport de recherche 3251, INRIA, September 1997.

[3] C. Duval, P. Erhard, C. Faure, and JC. Gilbert. Application of the automatic differentiation tool `odyssée` to a system of thermohydraulic equations. In J.-A. Désidéri, P. Le Tallec, E. Oñate, J. Périaux, and E. Stein, editors, *Proc. of ECCOMAS'96*, volume Numerical Methods in Engineering'96, pages 795–802. John Wiley & Sons, September 1996.

[4] F. Eyssette, JC. Gilbert, C. Faure, and JC Rostaing-Schmidt. Applicabilité de la différentiation automatique à un système d'équations aux dérivées partielles régissant les phénomènes thermohydrauliques dans un tube chauffant. Rapport de recherche HT-13/96/001/A, EDF/DER, February 1996.

[5] F. Eyssette, JC. Gilbert, C. Faure, and N. Rostaing-Schmidt. Applicabilité de la différentiation automatique à un système d'équations aux dérivées partielles régissant les phénomènes thermohydrauliques dans un tube chauffant. Rapport de recherche 2795, INRIA, February 1996. EDF/DER , HT-13/96/001/A.

[6] F. Eyssette, JC. Gilbert, C. Faure, and N. Rostaing-Schmidt. Applicabilité de la différentiation automatique à un système d'équations aux dérivées partielles régissant les phénomènes thermohydrauliques dans un tube chauffant. Rapport de recherche 2795, INRIA, February 1996.

[7] F. Eyssette and F. Vyskocil. Roundoff errors propagation analysis with automatic differentiation of fortran 77 code. In J.-A. Désidéri, P. Le Tallec, E. Oñate, J. Périaux, and E. Stein, editors, *Numerical Methods in Engineering'96*, pages 1018–1021. John Wiley & Sons, September 1996.

[8] C. Faure. Documentation succinte d'Odyssée version 1.6. Manuel d'utilisation, December 1996.

[9] C. Faure. Splitting of algebraic expressions for automatic differentiation. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation : Techniques, Applications, and Tools*, pages 117–127. SIAM, Philadelphia, Penn., 1996.

[10] C. Faure. Identification and optimization of complex models by sensitivity analysis. Development and Application of ODYSSEE an Adjoint Generator for Fortran Programs, Dresdes, June 1997.

[11] C. Faure. Template-driven automatic differentiation for large-scale scientific and engineering applications. Main features of Odyssee, IMA Special Workshop, Minneapolis, Minnesota, http://math.umn.edu/ santosa/adhome.html, July 1997.

[12] C. Faure. Third workshop on adjoint applications in dynamic meteorology. The cotangent mode in Odyssee,Bishop's University, Lennoxville, Quebec, Canada, http://www.tor.ec.gc.ca/adjoint/, June 1997.

[13] C Faure and C Duval. Application d'odyssée au logiciel de thermohydraulique THYC. Rapport de recherche HT-13/97/038/A, EDF/DER, December 1997.

[14] C. Faure and C. Duval. Automatic differentiation for sensitivity analysis. a test case. In K. Chan, S. Tarantola, and F. Campolongo, editors, *Proceedings of Second International Symposium on Sensitivity Analysis of Model Output (SAMO'98)*, volume EUR report 17758. EN, Luxembourg, 1998.

[15] C. Faure and Y. Papegay. Odyssée version 1.6. the language reference manual. Rapport technique 3251, INRIA, November 1997.

[16] A. Galligo and N. Rostaing. Une nouvelle technique de calcul formel appliquée à un problème d'optimisation en météorologie. In J.A Désidéri, L. Fezoui, B. Larrouturou, and B. Rousselet, editors, *Optimisation et Contrôle. Actes du colloque en l'honneur du soixantième anniversaire du professeur Jean Cea*, pages 143–160. cépaduès-éditions, 1993.

[17] A. Griewank and G.F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications.* SIAM, 1991.

[18] J. Grimm, L. Pottier, and N. Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In Berz, Bischof, Corliss, and Griewank, editors, *Computational Differentiation: Applications, Techniques, and Tools*, pages 95–106. SIAM, 1996.

[19] J. Grimm, L. Pottier, and N. Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. Rapport de recherche 2794, INRIA, February 1996.

[20] E. Hassold. Automatic differentiation applied to a nonsmooth optimization problem. In J.-A. Désidéri, P. Le Tallec, E. Oñate, J. Périaux, and E. Stein, editors, *Numerical Methods in Engineering'96*, pages 835–841. John Wiley & Sons, September 1996.

[21] J.-M. Malé, N. Rostaing-Schmidt, and N. Marco. Automatic differentiation: an application to optimum shape design in aeronautics. In J.-A. Désidéri, C. Hirsch, P. Le Tallec, E. Oñate, M Pandolfi, J Périaux, and E. Stein, editors, *Minisymposia of ECCOMAS 96*. John Wiley & Sons, Ltd, September 1996.

[22] B. Mohammadi, J.-M. Malé, and N. Rostaing-Schmidt. Automatic differentiation in direct and reverse modes: Application to optimum shapes design in fluid mechanics. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation : Techniques, Applications, and Tools*. SIAM, 1996.

[23] N. Rostaing. *Différentiation automatique: application à un problème d'optimisation en météorologie.* PhD thesis, Université de Nice-Sophia Antipolis, 1993.

[24] N. Rostaing and S. Dalmas. Automatic analysis and transformation of fortran programs using a typed functional language. Rapport de Recherche 1518, INRIA, 1991.

[25] N. Rostaing and S. Dalmas. Automatic analysis and transformation of FORTRAN programs using a typed functional language. In R. Glowinski, editor, *Proceeding of the 10th international conference on Computing methods in applied sciences and engineering*, pages 527–535. Nova Science, 1992.

[26] N. Rostaing, S. Dalmas, and A. Galligo. Automatic differentiation in odyssée. *Tellus*, 45A(5):558–568, 1993.

[27] N. Rostaing-Schmidt and E. Hassold. Basic functional representation of programs for automatic differentiation in the Odyssée system. In F.X. Le Dimet, editor, *Proceedings of the workshop on High Performance Computing in the Geosciences*, Les Houches (France), 1994. Kluwer Academic Publishers, NATO ASIE SERIES.

# Index

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803